Requirements, Design, and Development of a Rapidly Reconfigurable, Photo-Realistic, Virtual Cockpit Prototype

THESIS

Terry A. Adams
Captain, USAF

AFIT/GCS/ENG/96D-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/96D-02

Requirements, Design, and Development of a Rapidly Reconfigurable, Photo-Realistic, Virtual Cockpit Prototype

THESIS

Terry A. Adams
Captain, USAF

AFIT/GCS/ENG/96D-02

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

AFIT/GCS/ENG/96D-02

Requirements, Design, and Development of a Rapidly Reconfigurable, Photo-Realistic, Virtual Cockpit Prototype

THESIS

Presented to the faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial fulfillment of the Requirements for the Degree of

Master of Science in Computer Systems

Terry A. Adams, B.S.

Captain, USAF

December, 1996

Acknowledgments

First and foremost, I would like to thank Courtney, my wife, for all her love and understanding during my AFIT experience. She gave me the encouragement I needed to get through the long process of developing this thesis project. Without her taking care of our home and family, I could not have spent the many hours needed to complete my project. Any success during my AFIT stay is truly as much hers as it is mine. I would also like to thank my son, Nicholas, for all the times he ran to the door and shouted "I love you Daddy" after arriving home from another long day in the computer lab.

I would like to thank all the fellow Graphics Students for their help and assistance throughout the long process of completing our research. They let me bounce ideas off them and provided important feedback during this project's software development effort. I would also like to thank my fellow Thursday afternoon Wallyball players, for the fun and a great way to let off some steam.

Finally, I would like to thank my thesis committee, Lieutenant Colonel Stytz, Major Shomper, and Major Banks, for the guidance they provided during my Virtual Cockpit research. I appreciate the time and effort they put forth in reviewing and improving this thesis.

Table of Contents

List of Figures

List of Tables

Abstract

The United States Air Force uses aircraft flight simulators for pilot training and mission rehearsal. They use a variety of simulators for this task ranging with prices ranging from $400,000 to $30,000,000. These simulators have specialized hardware that restricts reuse of their components and increases maintenance costs. Air Education and Training Command wants to reduce simulators cost and improve availability to the operational commands by supporting research in virtual reality flight simulators.

This thesis looks at the development of a reconfigurable virtual cockpit in a distributed virtual environment that can be used for different aircraft as well as training scenarios. The thesis effort builds on a F-15 virtual cockpit previously developed at AFIT by creating a F-16. The Rapidly Reconfigurable Virtual Cockpit (RRVC) allows users to switch between an F-15 and F-16 during live simulation. All software models and aircraft geometry files are updated to reflect the current aircraft. The ability of a distributed virtual environment to support two unique aircraft flight simulators in a single application is encouraging. With the development of more aircraft, a single application can be provided to the operational pilot community that would support many aircraft at a fraction of the cost of today's flight simulators.

Requirements, Design, and Development of a Rapidly Reconfigurable, Photo-Realistic, Virtual Cockpit Prototype

1. Introduction

One of the United States Air Force's objectives is to maintain air superiority. The Air Force achieves this by recruiting and training quality pilots. The Air Force sends pilots through an extensive training program that teaches them how to fly military aircraft. Pilots then go through an additional training program for the specific aircraft they will be flying. The training period can last up to 3 years and the cost of this training can exceed $6,000,000. Pilots must continue to practice or they begin to forget the skills they have learned. Additionally, they must continue to learn new skills associated with a particular task or air combat mission. Unfortunately, practice is expensive and military budgets have been cut dramatically since the late 1980's. Simulators provide a less expensive way to train pilots then with conventional aircraft. Simulator's capabilities are constantly improving and offer an excellent tool for training. Unfortunately, the simulator's increase in reality has come at cost and not every unit can afford to have their own simulator. The overall goal of the Virtual Cockpit research is to develop an inexpensive aircraft simulator using conventional workstations, virtual reality, and distributed simulation.

1.1. Overview

Aircraft simulators have a wide range of capabilities and costs. An F-15 Weapons System Trainer (WST) costs the Air Force approximately $30,000,000 [OLSE96]. The F-15 WST is essentially a copy of the aircraft's cockpit and supports all the functionality of an operational F-15 aircraft. Each F-15 WST is specific not only to a model of aircraft, such as an F-15C; but, also specific to the block number of the aircraft (different versions of the same model). However, because of the large cost associated with this type of simulator the USAF only has a few WST's, which limits the amount of training that can take place on such systems. The sponsor for this research, Air Education and Training Command (AETC), is investigating ways to reduce the cost of simulators and increase their availability to pilots.

The rapid expansion in capabilities of graphics workstations has led to an explosion in modeling and simulation technology. Currently, the Air Force uses this technology to build simulators that allow a pilot to train in a particular aircraft. Known as Weapon Tactics Trainers (WTT) and Unit Training Devices (UTD), these simulators are good representations of a particular aircraft and cost anywhere from $400,000 to $800,000 [OLSE96]. Both the WTT and UTD contain limited out-the-window imagery, with only a small field of view displayed directly in front of the cockpit. With the arrival of virtual reality and distributed simulations, developing an aircraft simulation that can interact with other virtual aircraft all over the world in realistic scenarios is now possible [SCRI94].

The Air Force Institute of Technology's (AFIT) Virtual Cockpit (VC) is an ongoing research effort to produce a realistic, less expensive, virtual aircraft cockpit that can be use for pilot training. The VC's goal is to create a virtual reality simulation that can be used to train pilots in the use of aircraft displays and controls at a small fraction of the cost of a fully functional, WST simulator. Functionality in the VC, as described by Diaz, includes a low-fidelity aircraft dynamics model, photo-realistic cockpit displays for a single aircraft, out-the-window scenery, limited weapons' capabilities, throttle and stick input devices, and support for distributed simulations [DIAZ94]. The Virtual Cockpit's current configuration is an F-15, including aerodynamics, displays, sensors, and weapons. Development of the virtual cockpit's current

capabilities has been the product of several previous efforts [SWIT92] [ERIC93] [GERH93] [DIAZ94] [MCCA94] [SCHN95]. Additionally, work from both AFIT and Naval Postgraduate School provided both support and a framework for the development of the VC [COOK92] [SHEA92] [SNYD93] [KEST94]. The VC immerses the pilot into a virtual environment with a helmet-mounted display (HMD). The HMD not only allows pilots to look at their cockpit displays, it also allows them to view 360 degrees of imagery. The HMD provides an inexpensive alternative to million-dollar domed simulators, currently widely used to display such images [GUM94]. A reconfigurable VC will allow a single set of hardware and software to simulate many types of aircraft at a fraction of the cost to create one military training simulator. The VC research effort has progressed at a steady pace since inception. The research has shown that creating a realistic, low cost, virtual single-aircraft simulator is possible. This thesis research project focuses on expanding the capabilities of the Virtual Cockpit

1.2. Thesis Statement

The AFIT Virtual Cockpit can be reimplemented to allow rapid reconfiguration from one type of aircraft to another. Reconfiguration includes not only cockpit displays; but, also weapon systems, sensors, and aircraft aerodynamics. A F-16 Virtual Cockpit will be created to test rapid reconfiguration. Development of the system will focus on parameterized models to allow each type of virtual aircraft to share a common software model (i.e., one aircraft aerodynamic model can represent several different aircraft). The reconfigurable cockpit must support the current functionality of the current F-15 Virtual Cockpit and extend it to be able to represent a F-16. All F-16 cockpit models created will be photo-realistic to maintain the realism currently in the VC's F-15 cockpit. For weapons, VC will utilize a Virtual GPS receiver to steer a guided bomb to target. Finally, the Rapidly Reconfigurable Virtual Cockpit (RRVC) must support a distributed virtual environment.

1.3. Scope

Implementation of a reconfigurable cockpit is the primary goal of this research. An architecture will be created to allow a Virtual Cockpit aircraft to be rapidly reconfigured. The architecture is tested with two different virtual aircraft, F-15 and F-16. To exactly duplicate all functionality of an actual fighter cockpit is not the intent of this research. Instead the research focuses on reconfigurability and duplicates only the F-16's front instrument panel and a portion of its functionality. It is not the purpose of this research effort to design many different aircraft cockpits. The architecture is designed to support many different versions of aircraft and provide a framework for any future aircraft cockpit implementations. The input devices that are currently available in AFIT's Virtual Environments, 3D Medical Imaging, and Computer Graphics Laboratory, including aircraft throttle, stick, and rudder combination, magnetic head tracker, mouse, and keyboard provide all input and are used to simulate all aircraft controls. The VC research will use the Distributed Interactive Simulation standard to interface with other distributed virtual environments [IEEE93].

1.4. Assumptions

An assumption that impacts the realism of the simulation is that an F-16 stick and throttle can be adequately simulated using a stick and throttle based on the F-15. All F-16 stick and throttle inputs will be mapped to a F-15 counterpart.

1.5. Standards

The evaluation of the realism of a real-time, virtual environment is difficult. One standard will be to maintain an acceptable frame rate (the number of times the screen is redrawn per second). A frame rate of 15 frames per second will be considered acceptable performance. The Silicon Graphics Performer statistics tool will measure the frame rate of the simulation [MCLE92]. The goal of the effort is to increase the number of aircraft, the functionality of those aircraft, and provide the ability to rapidly reconfigure the

aircraft without affecting the current frame rate of 12-15 frames per second. In addition, the existing photo-realistic cockpit displays will be used as the standard for realism in the new aircraft cockpit that will be created. The entire application will be compared with the previous VC for performance in both the time to execute the model and the accuracy of the model. The simulation must also support the DIS standard [IEEE93].

1.6. Approach/Methodology

The development effort for a Rapidly Reconfigurable Virtual Cockpit can be broken down into four areas: system architecture, performance models, graphical models, and distributed simulation. Each of these areas will have their own distinct design and implementation issues. The area of primary importance to the entire development effort is the system architecture and initially most effort will be placed on developing a suitable system architecture.

1.6.1 System Architecture. The current VC's system architecture will be examined for suitability to meet this research effort's reconfigurability requirement. An architecture will then be designed that take the VC's current architecture into account while also considering the need to handle multiple aircraft. The F-15 currently implemented in the VC will be then transitioned to the new architecture. After transitioning the current VC, the architectures rapid reconfiguration ability will be tested using a derivative of the current F-15 (to represent a second aircraft), followed with testing using the newly developed F-16 VC.

1.6.2 Performance Models. All models will be examined for their ability to fit into the new architecture. The current VC's models will then be redesigned if necessary to match the new architecture. In an effort to increase the current VC fidelity, a software model for aircraft aerodynamics will be acquired from the Wright Laboratory's Control Integration and Assessment Branch (WL/FIGD). This model has the capability to represent many different aircraft. The model will be integrated into the new architecture and tested for several aircraft. The weapon systems removed from the latest implementation of the VC will be reintegrated into the aircraft. All of the current and new models will also be made to allow real-time reconfigurability for different aircraft configurations.

1.6.3 Graphical Models. Current F-16 simulators and the F-16 System Program Office (SPO) will be contacted to check availability of any existing models. If no acceptable graphics models of the F-16 exist then new models will be developed using Coryphaeus' Designers Work Bench. Realism will be of utmost importance and comparison with actual F-16 aircraft will be accomplished to ensure accuracy of graphical models

1.6.4 Distributed Simulation. The World State Manager 3.0, developed by Steven Sheasby for AFIT, will be integrated into the VC. World State Manager 3.0 provides a communication layer between Graphics Lab applications and the current Distributed Simulation environment, currently DIS. All communication between the application and network entities will be accomplished using this communication World State Manager 3.0 will support all necessary DIS PDUs.

1.7. Materials and Equipment

The main piece of equipment needed for this thesis effort is a four-processor Silicon Graphics Onyx with Reality Engine 2 graphics. Software for the system must include a C++ compiler, Performer 2.0, and Coryphaeus' Designers Work Bench (DWB) version 3.1. A Thrust Master throttle and stick combination is required to fly the virtual cockpit. The VC can be viewed through a conventional computer monitor; however, to enhance the realism a Polhemus Laboratories head-mounted display and a Polhemus magnetic head tracker is required. All equipment needed for this effort is currently available in AFIT's Virtual Environments, 3D Medical Imaging, and Computer Graphics Lab.

1.8. Thesis Organization

The six chapters and two appendices in this thesis provide a background of relevant research areas and a description of the analysis, design, and development of a rapidly reconfigurable virtual cockpit. Chapter 2 provides an overview of relevant research including virtual reality, flight simulators, software architectures, distributed simulation, and reconfigurable cockpits. Chapter 3 details the requirements and the top-level design of the RRVC. Chapter 4 is an overview of the implementation of the design. Chapter 5

is a discussion of results for the project related to each of the individual project requirements. Finally, Chapter 6 provides personal conclusions on the research along with a recommendation of areas for future research. Appendix A consists of several photos of the F-16 VC's cockpit models. Appendix B contains source code for the Common Object DataBase (CODB), which also contain examples of CODB use.

2. Background

This chapter will explore areas of research related to the development of a rapidly-reconfigurable virtual cockpit. Technologies that support this project are virtual reality, flight simulators, distributed simulations, modeling, and software architectures. The chapter provides a description of these areas along with current research efforts that are pertinent to this thesis effort. Since the Virtual Cockpit has been ongoing development in the Graphics Laboratory, the chapter provides a short history of the project.

2.1. Virtual Reality

Considerable knowledge currently exists about virtual reality environments, with many diverse types of applications completed and under development. "Virtual Reality is a new type of human-computer interface that aims at the user the illusion of being immersed in a computer generated reality providing a more direct communication link between users and the problem environment modeled by the computer system [FIGU93]." Virtual reality was new in 1993 when Figueiredo wrote the article. However, virtual reality applications exist in a wide range of fields today, from architecture and baseball batting practice to training radiotherapists and astronauts [CATE95][ANDE93] [ASTH93][MOSH86]. Virtual reality systems use display devices and input devices that allow natural interaction with the virtual environment. Display devices include computer monitors, head-mounted displays, head-coupled displays and various projection systems [BOLA93]. Input devices range from the basic keyboard and mouse, to eye, head, and hand trackers [STUR94]. Virtual reality systems must provide realistic image quality in real time and realistic modeling of the various entities in the virtual environment. If graphics and models are unrealistic then users will not feel as if they are actually participating in the virtual world, that is a critical component of training.

One interesting virtual environment software application by Andersson is the Virtual Batting Cage [ANDE93]. This application has two research areas in common with the Virtual Cockpit, virtual reality and training systems. The Virtual Batting Cage is a virtual environment where a batter uses a real bat to swing at a virtual ball thrown by a virtual pitcher. The Virtual Batting Cage uses a software model to model the ball

and a videotaped image of a pitcher to enhance realism. Both a head-mounted display and the real baseball bat use positional trackers to evaluate the batter's motion. A batting sequence begins with the batter looking out at the pitcher on the mound, the videotaped pitcher then "pitches" the ball, the batter swings at the pitch, and finally the system shows whether the batter hit the ball or missed the ball. The computer accomplishes this by tracking the bat and determining the point of impact with the computer-modeled ball. The batter can then replay the at-bat, stop the replay at any time, look at ball and bat position, and identify any mistakes that they made. The system allows situations and pitches that would be extremely difficult to attain without a virtual environment. Andersson uses a real bat to increase the experience's realism, because no simulated input device could accurately duplicate a bat. Andersson stresses the significance of real-game situations for training purposes and the importance of accurate models in a virtual reality environment. This application has many conditions that are similar those associated with training fighter pilots including the following:

1. the focus on training in game-like situations (realism) without risk to participants,
2. importance of split-second decisions (real-time performance) in training,
3. the need for realistic input devices, and
4. the need for off-line training analysis in both real-time and slow-motion.

Training a pilot and training a batter are similar in that they are both engaged in dangerous (Condition 1) highly stressful, split second decision making (Condition 2). For training to be realistic, the interactions between trainee and simulation must be realistic (Condition 3) and the simulation environment must be realistic. When undertaking training in rapidly changing situations (100 pitches or a 2 hour combat mission), an important part of the training experience is the ability to review what took place and the trainee's reaction. Since the batting trainer and a virtual cockpit have similar design consideration, the design of the reconfigurable VC will take the batting trainer's considerations into account.

AFIT's Graphics Laboratory is involved in Virtual Reality research and has been for many years. Students have developed projects such as a Virtual Emergency Room [GARC96] and a Satellite Modeler [WILL96]. Most of the projects utilize a Head Mounted Display to immerse the user into the application's

particular computer-generated environment. A head tracker uses the information on where the user is looking and then the application displays only that portion of the computer generated world. Interaction with the environment is accomplished via a hand tracker, keyboard, or mouse. The Virtual Cockpit is itself one of the AFIT's Graphics Laboratory's projects and makes use of many of the same input devices; however, to enhance the flying experience the VC integrates an aircraft throttle, stick, and rudder device into the environment as an input device. This input device allows users to interact with the virtual airplane in the same manner as they would with a real airplane, with a stick, a throttle, and rudders. The stick, throttle,and rudders give pilots the same input device they would have in the cockpit, increasing the realism of the virtual environment.

2.2. Flight Simulators

Training is an important part of maintaining operational readiness in the Air Force. The Air Force widely uses flight simulators as a valuable part of pilot training. Simulators are used for all types of training from initial pilot training to combat mission rehearsal. The flexibility of simulators allows them to represent many diverse and dangerous environments without risk to the pilot. Aircraft simulators have a wide range of capabilities and costs as can be seen from the following descriptions.

2.2.1 Weapons System Trainer. As mentioned in Chapter 1, the WST is the Air Force's highest fidelity simulator provides all the functionality of an operational F-15 aircraft. While the functionality of the cockpit, is identical to an operational F-15, the simulator does have limitations. Even with the great cost associated with the WST, the system has limited visual cues. To give the feeling of movement, the WST utilizes a front projection screen, driven by a Silicon Graphics computer, that displays earth and sky imagery. The display updates the imagery to reflect the aircraft's current orientation and position in the world or scene. However, this type of display has limitations because it only allows the individual in the aircraft's front seat to see what is "outside" of the windscreen (the outside environment is generated by the computer) and only for the small field of view that can be represented with a large screen monitor. While, the WST has a few limitations such as no motion and limited visual cue, it is the closest thing the Air Force has to actually flying in a real F-15, while still being on the ground.

2.2.2 Weapons Tactics Trainers (WTT) and Unit Training Devices (UTD). The huge expense of WST's has caused the Air Force to look for less expensive alternatives that will still provide realistic training. At the same time, a rapid increase in capabilities of graphics workstations has led to an explosion in modeling and simulation technology. Workstations provide a less expensive way to build flight simulators then the specialized hardware previously required [SWIT92]. Workstations are used to build simulators that allow a pilot to train in a particular aircraft. Known as Weapon Tactics Trainers (WTT) and Unit Training Devices (UTD), these simulators are medium fidelity representations of a particular aircraft and have a hardware cost anywhere from $400,000 for the WTT to $800,000 for the UTD (Note: the cost of the software is considered a one-time cost and is not included in these costs)[OLSE96]. The WTT and UTD are considered medium fidelity simulators because they do not have all the functionality, in both cockpit controls and aircraft modeling (i.e., landing gear), associated with an actual operational aircraft. The primary difference between the WTT and the UTD are the training purpose and aircraft controls. The UTD is a general purpose training simulator, while the WTT's intent is to train the pilot in a particular function or aircraft area such as the radar. As a general training device, the UTD has most of the aircraft's controls, including those switches and dials on the aircraft's left and right consoles. The WTT, on the other hand, only has the switches and dials required for its specific training task. Both the WTT and UTD contain limited out-the-window imagery, with only a small field of view displayed directly in front of the cockpit. The VC is part of the effort to provide low-cost training with the realism necessary to be effective.

2.2.3 Domed Simulators. Domed simulators were created in an effort to increase the realism of flight simulators. Domed simulators are located inside a sphere as seen in Figure 2-1. Figure 2-2 depicts a common configuration in which a cockpit is placed in the center of the dome and a computer generated image of the pilot's out-the-window view is displayed on the inside of the dome. The dome can provide up to 360 degrees of imagery for the pilot, giving the pilot almost the same field of view as that in an actual aircraft. The cost of the domed simulator shown is approximately $30 million, including the cost of the dome structure and the computer equipment necessary to generate the visual displays. The Large Amplitude, Multimode Aerospace Research Simulator (LAMARS) in Figure 2-3 incorporates motion into

domed simulators. The LAMARS has five degrees of freedom and can simulate motion up to the limits of its 30 foot arm. The Flight Simulation Facility in Bldg. 146 at Wright-Patterson Air Force Base contains both simulators shown. It is important to note that many simulations that take place in the facility do not use the motion-based dome because motion does not provide a large increase in realism. Engineers in the facility have found that graphics displayed to pilots provide the necessary motion cues. This fact is of great interest to the Virtual Cockpit research, because the VC will also have 360 degrees of imagery; but, will have no other motion cues. The Virtual Cockpit will provide similar visual cues to the pilot at a fraction of the cost. Therefore, the domed simulator provides a good comparison for the VC.

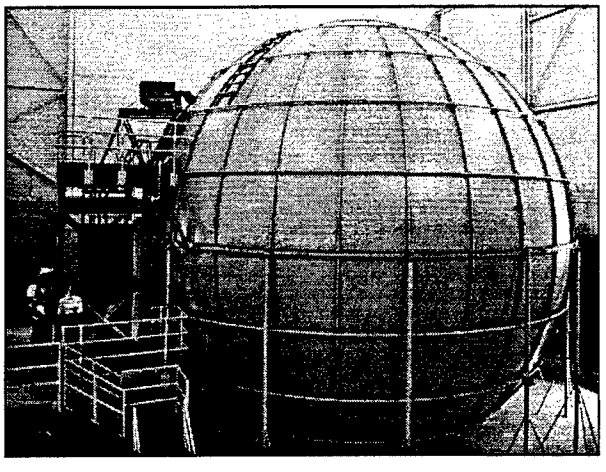

Figure 2-1. Exterior of Dome Flight Simulator

Figure 2-2. Interior of Dome Flight Simulator

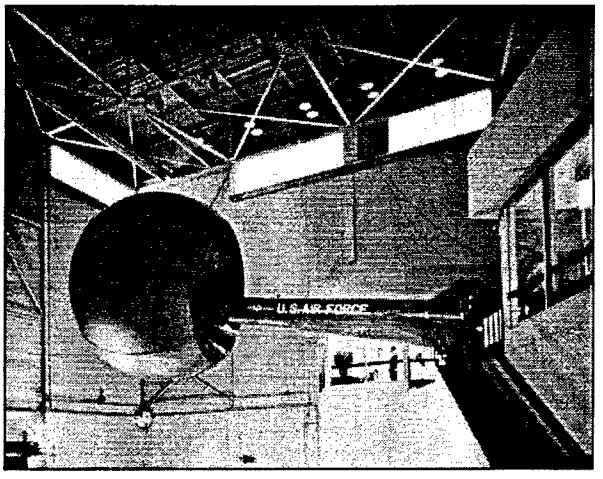

Figure 2-3 LAMARS Flight Simulator

2.3. Distributed Simulation

Distributed simulation is the ability to support interaction between multiple participants at geographically separate sites using a network. Currently, the two primary military distributed simulation architectures are Simulator Networking (SIMNET) and Distributed Interactive Simulation (DIS). A third architecture, the High Level Architecture (HLA), is currently undergoing definition. SIMNET was one of the first projects to standardize a wide-area network protocol. SIMNET was used to develop simulations using hundreds of networked simulators; but, it is not widely used today [MCCA94]. DIS is an extension of SIMNET and is the network simulation protocol currently being used in most distributed military simulations. DIS uses a highly standardized format in which all simulators on the network must communicate using standard messages called Protocol Data Units (PDU) [MCDO91]. The HLA architecture improves on DIS by creating additional message types and allowing more freedom in the type of communication that may take place between simulators [STAR96]. HLA is currently being developed by the Defense Modeling and Simulation Office with a projected release date of December 1996.

2.3.1 Distributed Interactive Simulation. The DIS Standard defines distributed interactive simulation as follows: "DIS is a time and space coherent synthetic representation of world environments designed for

linking the interactive, free play activities of people in operational exercises[IEEE93]." The following concepts provide the basis for the DIS architecture:

1. No central computer controls the entire simulation exercise
2. Autonomous simulation applications are responsible for maintaining the state of one or more simulation entities
3. A standard protocol is used for communicating "ground truth" data
4. Changes in the state of an entity are communicated by simulation applications
5. Perception of events or other entities is determined by the receiving application
6. Dead-reckoning algorithms are used to reduce communications processing. [IEEE93]

DIS is a broadcast protocol that sends packets to all users on the network, because all simulation applications are responsible for determining the state of the simulation (concept 1). Network entities must listen to all broadcast data to determine what pieces of information are of interest to them.

2.3.1.1. Data Formats. The standard sends data in a format referred to as a Protocol Data Units (PDU). Each PDU has a fixed size and format header that includes the following information: protocol version, DIS exercise identification, type and family of PDU, time stamp, and length of PDU. This information allows entities on the network to determine items of interest in a particular PDU that has been received. For instance, if a DIS exercise identification is not consistent with that entity's identification then the application can ignore the PDU. The DIS Standard provides 27 types of PDU's that a DIS-compliant simulation uses, see Table 2-2 (DIS PDU Families and Types). An Entity State PDU is the PDU used to update an entity's position in a simulation, see Table 2-1 (Entity State PDU) [IEEE93].

Table 2-1 Entity State PDU Format

Field Size (bits)	PDU Section	PDU Fields
96	PDU Header	Protocol Version - 8 bit enumeration
		Exercise ID - 8 bit unsigned integer
		PDU Type - 8 bit enumeration
		Protocol Family - 8 bit enumeration
		Time Stamp - 32 bit unsigned integer
		Length - 16 bit unsigned integer
		Padding - 16 bits unused
48	Entity ID	Site - 16 bit unsigned integer
		Application - 16 -bit unsigned integer
		Entity - 16 bit unsigned integer
		8 bit enumeration
8	Force ID	8 bit unsigned integer
8	# of Articulation Parameters (n)	Entity Kind - 8 bit enumeration
64	Entity Type	Domain - 8 bit enumeration
		Country - 16 bit enumeration
		Subcategory - 8 bit enumeration
		Specific - 8 bit enumeration
		Extra - 8 bit enumeration
64	Alternative Entity Type	Entity Kind - 8 bit enumeration
		Domain - 8 bit enumeration
		Country - 16 bit enumeration
		Subcategory - 8 bit enumeration
		Specific - 8 bit enumeration
		Extra - 8 bit enumeration
96	Entity Linear Velocity	X Component - 32 bit floating point
		Y Component - 32 bit floating point
		Z Component - 32 bit floating point
192	Entity Location	X Component - 64 bit floating point
		Y Component - 64 bit floating point
		Z Component - 64 bit floating point
96	Entity Orientation	Psi - 32 bit floating point
		Theta - 32 bit floating point
		Phi - 32 bit floating point
32	Entity Appearance	32 bit record of enumerations
320	Dead Reckoning Parameters	Dead Reckoning Algorithm - 8 bit enumeration
		Other Parameters - 120 bits unused
		Entity Linear Acceleration - 3x32 bit floating point
		Entity Angular Velocity - 3x32 bit floating point
96	Entity Marking	Character set - 8 bit enumeration
		11 8-bit unsigned integer
32	Capabilities	32 Boolean fields
n x 128	Articulation Parameters	Parameter Type Designator - 8 bit enumeration
		Change - 8 bit unsigned integer
		ID - attached to - 16 bit unsigned integer
		Parameter type - 32 bit parameter type record
		Parameter value - 64 bit

Table 2-2. DIS PDU Families and Types.

Entity Information Interaction	Simulation Management	Emission Regeneration
Entity State	Start/Resume	Electromagnetic Emission
Collision	Stop/Freeze	Designator
	Acknowledge	
Warfare	Action Request	Radio Communications
Fire	Action Response	Transmitter
Detonation	Data Query	Signal
	Set Data	Receiver
Logistics	Data	
Service Request	Event Report	
Resupply Offer	Message	
Resupply Received	Create Entity	
Resupply Cancel	Remove Entity	
Repair Complete		
Repair Response		

2.3.1.2. Dead Reckoning. Because of the large amount of data that a DIS exercise sends out, even a small exercise can overwhelm a network. However, this does not eliminate the importance of accurate positional data in conducting a realistic combat simulation [SCRI94]. One technique to reduce the number of packets broadcast, while still maintaining accurate positional data for entities in the simulation, is dead-reckoning. Dead reckoning reduces the amount of network bandwidth needed to pass state information across the network by calculating a vehicle's future position based on its past state. When a network entity receives an entity state PDU from another entity (sender), the receiving entity updates the current position of that entity (receiver). It then begins using the velocity information in the PDU to update that entity's position. The sender also performs the same calculations. When the sender determines that the calculated position is not within a predetermined threshold of the actual position, it sends out a new entity state PDU, with its actual position, to everyone on the network. The receiver then uses this information to update the entity's state in their simulation. This greatly reduces the traffic on the network [HARV91]. One disadvantage of this approach is that if an entity continues along a straight path (never changing state) it would never send out another entity state PDU after sending its initial one. Any new entities that then come on-line would not know about this straight-path entity. To eliminate this problem, the standard dictates that each entity in a simulation will send out a PDU at least once every five seconds [IEEE93]. Dead-reckoning

is an important part of distributed simulation. Both the military and civilian sectors widely use dead-reckoning to reduce network traffic.

It was originally assumed that dead-reckoning would not be useful for fighter aircraft because of their rapid state changes (position, orientation, velocities, and accelerations). Harvey and Schaffer refute the claim that dead reckoning can only be accurate for slow-moving vehicles such as tanks or armored personnel. They used first-order dead-reckoning (using velocity to compute future vehicle position) and several allowable error thresholds in their testing [HARV91]. The results found that first order dead reckoning produces highly accurate position and orientation information [HARV91]. In addition network traffic can be reduced from 100 packets per second to less than 20 packets per second, a reduction of over 80% [HARV91].

Singhal and Cheriton propose an alternative method to reduce network traffic using a history-based protocol [SING95]. Their proposed solution is similar to dead-reckoning in that it uses past data about the aircraft to predict future position; however, it uses the information differently. Instead of predicting future positions using velocities and accelerations, the authors use only position information. Their algorithm performs a curve-fit analysis based on position to decide where the entity will be and when to broadcast new state updates. This protocol has an advantage over dead-reckoning because it only requires network to send out position information over the network, reducing the amount of bandwidth needed for state updates by 66%. Dead-reckoning requires the network to broadcast position, velocity, and sometimes acceleration data. In traditional dead-reckoning, when a simulator receives a new entity state PDU, all simulations on the network update the position of that entity to its newly received true position. However, the curve-fitting algorithm's converges the entity's new state with the path calculated in the curve-fit approach. This reduces the amount of jumpiness that occurs in traditional dead-reckoning upon receipt of a new entity state PDU. The authors found that dead-reckoning outperformed their history-based protocol by 25% when considering average error. However, history-based inputs can be sent out 1.75 times more than dead-reckoning/DIS updates and still reduce bandwidth. When considering this increased update rate, the

history-based approach outperforms dead-reckoning. A more accurate dead-reckoning approach could improve the accuracy and realism of all DIS simulations, including the VC.

2.3.2. High Level Architecture. Stark, Weatherly, and Wilson provide an overview of a new form of architecture currently under development for distributed simulations, the High Level Architecture (HLA) [STAR96]. HLA is to replace the Distributed Interactive Simulation (DIS) standard. While DIS architectures communicate with Protocol Data Units (packets of data broadcast over the network), HLA simulations communicate to the simulation environment through a common software module. The HLA Runtime Infrastructure (RTI) is the common software module and it provides a set of common services. The services consist of Federation Management, Time Management, Declaration Management, Ownership Management, and Object Management. Instead of broadcasting data like DIS, HLA sets up federations in which producers and receivers of simulation data are registered for a particular group of entities, known as a federation. These federations are groups of distributed simulations that wish to conduct an exercise. An HLA simulation communicates to the RTI through the RTI Application Programmer's Interface (API). In order for a simulator to communicate within HLA, it must compile the API into its software and use the RTI services to communicate [STAR96]. The HLA architecture eliminates some commonly noted deficiencies of DIS, such as excessive PDU sizes, inflexible PDU formats, and a broadcast only architecture [MACE94]. This is of particular interest to the VC because it must move from the DIS standard to the HLA standard. Additionally, the flexibility of HLA in data exchange could open the VC to many other uses. [STAR96]

2.4. Software Architectures

Another component of this research involves software architectures that tie together the areas of distributed simulations, virtual environments, and flight simulations. While SIMNET, DIS, and HLA are concerned with information between simulators, a software architecture is concerned with how software components communicate in a single application. Two basic approaches stated by Amselem for communication infrastructures, a peer-to-peer configuration and a centralized configuration, shown in Figure 2-4. A star configuration allows peer-to-peer communication among all of the software components in the system, increasing the software's complexity. A centralized configuration allows communication via

a centralized component that stores information that other components may need [AMSE95]. Most virtual environment designers employ one of these configurations when developing an application. The object-oriented development techniques encourage encapsulation of data and will support either approach. However, a centralized configuration is more in line with the object-oriented methodologies of abstraction and encapsulation [RUMB91].

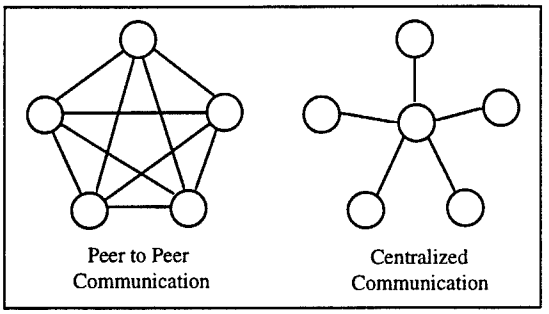

Figure 2-4. Communication Approaches.

2.4.1. Peer-to-Peer Communication. The AFIT Graphics Laboratory currently has two software architectures in place. The first one is ObjectSim an architecture that allows virtual reality developers to construct distributed simulations quickly [SNYD93]. The ObjectSim architecture is very tightly coupled to Silicon Graphics' Performer package and allows efficient multiprocessor rendering [STYT95]. ObjectSim provides a set of classes that serve as base classes for creating a simulation. The classes support such concepts as multiple players (both DIS and non-DIS), multiple viewpoints, terrain, and input devices [SNYD93]. However, in some cases the library does not provide a class to support a specific requirement or abstraction. The developer modifies one of ObjectSim's abstract classes to fit the requirement. The ObjectSim architecture also makes heavy use of peer-to-peer communication, although much of it is hidden from the user, through the use of tightly coupled classes. ObjectSim is a powerful tool that will allow an individual to quickly develop a distributed, virtual environment application.

2.4.2. Centralized Communication. A second approach, being utilized in the AFIT Graphics Laboratory is the Common Object DataBase (CODB) architecture. Stytz proposes a centralized software architecture

using a database to store information [STYT97]. The CODB uses an object-oriented design to allow a software developer to easily create a centralized database for storing any type of simulation data. The database employs a double-buffering scheme that allows simulation components to read and write simultaneously. Additionally, the architecture uses shared memory to store the data, that provides an efficient data access scheme for multiple process applications. Because the system utilizes shared memory and double-buffering, it also allows processes running at different rates to communicate without excessive blocking. In this architecture, the developer creates simulation components along with the database structure that they will update. This allows developers to abstract away from class access methods and use a common access method to read any of the shared simulation data. Any class can access the data placed in the database by any other class. The CODB accomplishes this by providing certain access methods, see Rumbaugh diagram in Figure 2-5. Currently, simulation components / classes have been created that support several input devices, a simple Performer renderer, several aircraft models, a DIS interface, and a generic entity generator for testing purposes. The CODB architecture's primary concern is with communication of data within an architecture and does not contain a comprehensive set of simulation creation functions. A sample simulation developed for testing included several basic simulation classes that would be common to many simulations. As more CODB development is accomplished, more modules will be available for use, and simulations can be developed more quickly. In conclusion, the CODB architecture does not support all of ObjectSim's features; however, it does provide increased flexibility in creating distributed, virtual environment applications [STYT97].

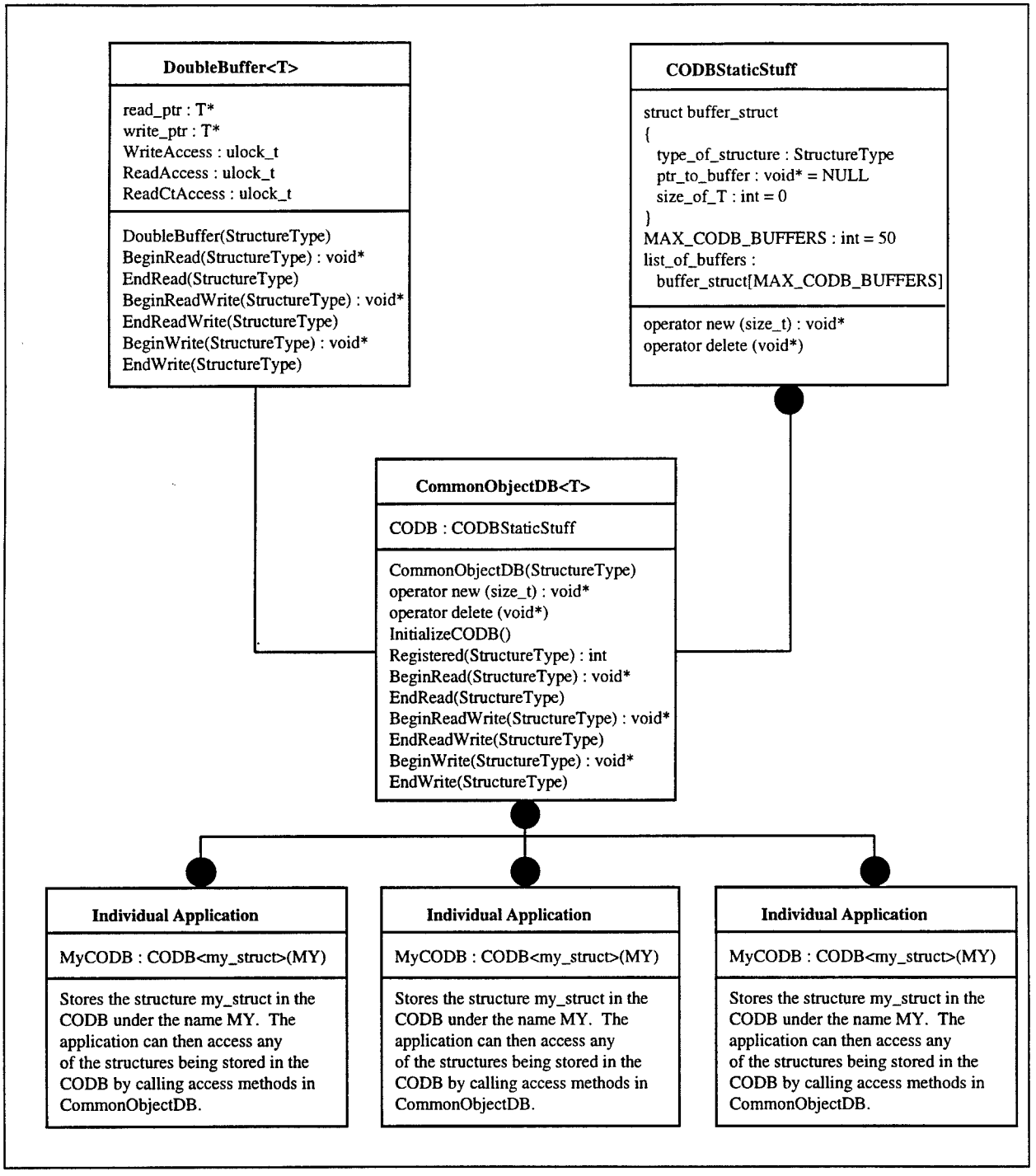

Figure 2-5. CODB Rumbaugh Diagram

2.4.3. Mixed Model Communication. The Distributed Interactive Virtual Environments (DIVE) is a well-known, long-term effort primarily concerned with ways that distributed virtual realities communicate and interact [HAGS96]. DIVE is different from most architectures in that it uses both peer-to-peer communication and a centralized database. DIVE uses a relatively simple replicated database to store information about all the components in the world. Conceptually, it is called a shared database; however in the distributed environment it is really several copies of a single database. The DIVE distributes the database from peer-to-peer [CARL93]. This database provides information about all objects in the world. For instance, if a person in the virtual reality turns around, they update their database and then send out this information to a server that sends it to the rest of the systems on the network [HAGS96]. Mutually exclusive locks insure that only a single process is accessing data on a local database or updating an object in all the distributed databases in the environment [CARL93]. The concept employed by DIVE in network communication could be employed in an individual software application by creating a data structure for each software component to store its information. The information would be then passed to all other software components. This is similar to the CODB approach; however, all of the simulation components' shared data remains in a centralized database..

An architecture that does not appear to fit the mold of a centralized / decentralized architecture is the MR toolkit presented by Shaw [SHAW92]. Shaw proposes a decoupled architecture where all simulation components act independently of each other. The architecture proposed has four distinct components for a virtual reality simulation; these include Interaction, Computation, Geometric Model, and Presentation. The Computation component can execute independently and communicate asynchronously with the other components, hence the decoupled designation. By having a separate computation component a model can execute continuously without being tied to the input device's update cycle (interaction component) or the output device (presentation component). The components all act independently; however, they all can communicate via a common package. The MR package's component that facilitates communication between components or processes is the Data Sharing package. The Data Sharing package is essentially the shared component of the centralized configuration mentioned above. The package

provides the programmer with a high level interface to communications between processes without having to worry about sockets and network connections. This level of abstraction allows programmers who are not expert networkers to create network simulations. MR is a software architecture that is extremely powerful because it allows a single application to be placed on multiple processes or multiple machines. The Data Sharing package is a centralized approach that is applicable to the Virtual Cockpit.

2.5. Physical Modeling

Physical modeling is the simulation of physical geometry and physical effects on a computer. This section will only cover these areas as they pertain to the VC and its operating environment, primarily a Silicon Graphics Computer System. The VC will utilize modeling tools widely used in the AFIT's Graphics Lab. The primary tools available for modeling physical geometry are Coryphaeus' Designer's Workbench (DWB) and MultiGen. As for modeling physical effects, Silicon Graphics has developed a software development package called Performer. Performer consists of several libraries of software routines for high-performance graphics applications [MCLE92] [ROHL94]. The virtual reality and computer graphics communities both widely use Performer for modeling virtual worlds [CARL93] [GIVE 95] [MACE94] [ROY94] [STYT95].

2.5.1. Physical Geometry. DWB and MultiGen provide extensive capabilities including geometry design, materials, lighting effects, texture mapping, and multiple levels-of-detail for a single model. For more information on the functionality of the individual programs see MultiGen Modeler's Guide [MULT94] and Designer's Workbench 3.1 Reference [MILB95].

2.5.2. Physical Effects. Performer provides several libraries for modeling many of the physical effects that can change the way a model looks, including: position, orientation, geometrical transformations (scale, rotate, shear), light, and even fog. Performer reads many common modeling packages' output, including MultiGen and DWB. The Performer library, *libpf*, allows users to create a scene that contains physical geometry. The physical geometry takes the shape of a hierarchy that consists of nodes connected in an acyclic graph. Each of these nodes contains part of the scene's geometry. Figure 2-6 lists the hierarchy of Performer's many node types, that include pfGeodes (one or more pfGeoSets), pfBillboard (automatically

rotates pfGeoSets to face eyepoint), pfLightPoint (visible but non-illuminating points of light), and pfLightSource (non-visible but illuminating light source) [ROHL94]. Nodes also exist that can transform or group together other types of nodes. The scene hierarchy generally takes the shape of a tree with all top-level nodes influencing all nodes below it in the tree. After the developer creates the geometry tree, Performer will automatically traverse the tree, culling parts of the tree that are not visible and drawing parts of the tree that are visible. The library also allows the user to easily create multiple processor graphics applications with all of the memory management and system calls hidden from the user. Performer is an extremely large and powerful library of functions that gives the developer great freedom in developing real-time graphics applications. This is a limited description of the hundreds of functions available, for more information refer to the *IRIS Performer Programming Guide* [MCLE92].

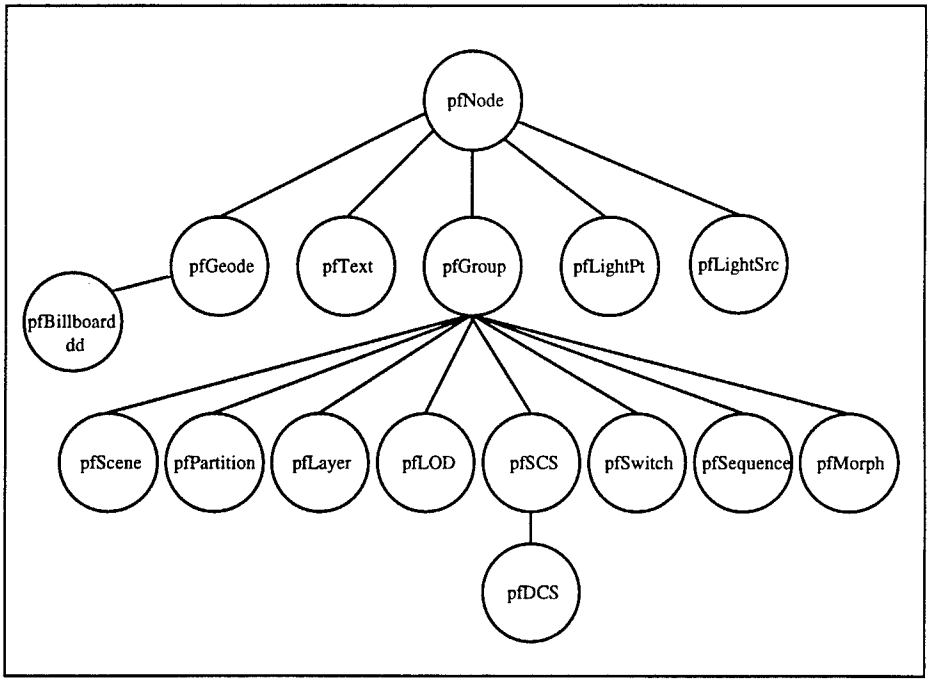

Figure 2-6. Performer Node Hierarchy.

2.6. Aircraft Simulator Reconfigurability Research

Two approaches have been taken in reconfigurability research: cockpit instrumentation reconfiguration and reconfigurable software simulation components. In reconfigurable cockpit instrumentation, software reconfigures the instrumentation based on the type of aircraft being flown. The Federal Aviation Administration's (FAA) Reconfigurable Cockpit Simulator (RCS) simulates most of the commercial transport aircraft. The RCS uses a network of five Silicon Graphics computers to model aircraft aerodynamics, the flight management system, drive cockpit displays, and approximately 90 percent of the aircraft's other major functions [FAA95]. The FAA uses the RCS for Human Factors evaluations [FAA95]. Wright Laboratory's Panoramic Cockpit Control and Display System (PCCADS) uses large screen monitor to display the entire instrument panel to its pilots. Pilots interact with the system's instrument panel with by touch screen. The PCCADS updates the controls and displays based on the portion of the screen touched by the pilot. Wright Laboratory uses PCCADS to develop advanced control and display techniques to increase pilot's situational awareness and safety [WL95]. Both of the cockpits have a limited out-the-window display capability with a small field-of-view. RCS and PCCADS both allow pilots to fly several different types of aircraft cockpits from a single control station.

The other part of the reconfigurablity equation is reconfigurable software simulation components. Flight simulation components are software modules that model a portion of the aircraft's functionality. For instance, an aircraft's aerodynamic model is based on the same laws of motion regardless of aircraft type. However, different aircraft types have attributes that affect their performance. A reconfigurable software model attempts to encapsulate common features among all aircraft into a single location and allow differences to be parameter driven based on aircraft type. One such model is an aircraft aerodynamic model developed by Eidetic's for WL/FIGD to represent many different types of threat aircraft. The design of the Eidetic's model is consistent with other reconfigurable models and will be described as a representative reconfigurable model. The Eidetic's aerodynamic model encapsulates all the aircraft's laws of motion into a single model. The type of aircraft to be simulated initializes the model. The aerodynamics model uses the aircraft type information to read data files that describe that aircraft's performance. The data includes

tables for coefficients of lift, coefficients of drag, and thrust for different aircraft speeds and altitudes. The model then uses the equations of motion and aircraft data to model the performance of that particular aircraft. Other aircraft can be modeled by creating data tables for those aircraft. Reconfigurable software simulation components provide a valuable tool in developing reconfigurable simulators. The Eidetic's model discussed above is a library of C functions that will be reused in the VC project for aerodynamic modeling. Knowledge of other reconfigurability projects is important not only from a reuse standpoint; but, to also gain insight into the techniques other simulation developers have used to make simulators reconfigurable.

2.7. Virtual Cockpit

The Virtual Cockpit was originally developed as a low-cost distributed virtual environment for pilot training. The Virtual Cockpit was developed by using Performer on a Silicon Graphics workstation. The distributed environment was based on the DIS standard. Switzer started the project in 1992 and furthered the overall idea of a virtual flight simulator and developed software to interface with the various input devices that the simulator would employ [SWIT92]. Also in 1992, McCarty developed an out-the-window display for the Virtual Cockpit [MCCA94]. The VC research project was the basis for two student's theses in 1993. Gerhard developed low-fidelity weapon systems for the VC, including a weapons controller [GERH93]. Classes were developed for a cannon, various types of bombs, and missiles. Erichson developed a sensor suite for the VC. The sensors developed included a radar model, infrared model, and an Inertial Navigation System (INS). The sensor development also included displays for the various sensors, although the infrared display utilized Performer functions that prevent it from being useful in a VR application [ERIC93]. In 1994, Diaz created a photo-realistic F-15cockpit that allowed a pilot to interact with switches and dials via the mouse [DIAZ94][KEST94]. The VC Project continued in 1995, with Schneider developing the ability to switch the VC between human control and an artificially intelligent wingman developed by Edwards [EDWA95] [SCHN95]. The VC continues into 1996 with the ability to represent different aircraft and rapidly switch between them.

2.8. Conclusion

A wealth of knowledge exists in each of the fields of study needed to develop a Rapidly-Reconfigurable Virtual Cockpit (RRVC). Virtual reality applications are becoming more widely used for a variety of applications from batting practice to training astronauts. Flight simulators are available in a wide range of fidelities and capabilities. However a narrow field-view for the pilot's out-the-window display inhibits many of these very expensive flight simulators. Computer modeling and software architectures provide a framework to develop flight simulators. Computer modeling provides flight simulators with software models of aircraft dynamics and functionality, in addition to geometrical models to represent an aircraft's structure or cockpit. A software architecture provides a framework for a software application regarding both structure of the software components as well as communication between components. Distributed simulation provides a way to link several flight simulators or other entities and allow them to interact together. Distributed simulation is similar to a software architecture on a networking level. The Rapidly-Reconfigurable Cockpit's design utilizes concepts and ideas from each of the disciplines discussed above to create a reconfigurable aircraft cockpit in a distributed virtual environment.

3. Requirements and Design

Chapter 3 discusses both the requirements and design needed to complete a Rapidly Reconfigurable Virtual Cockpit (RRVC). The primary requirement for this effort is reconfigurability of the VC. Requirements that fall under the reconfigurability requirement include the following:

- a computer architecture that supports reconfigurability,
- software models that support reconfigurability (aeronautical, sensor, weapons),
- a F-16 cockpit, to complement the F-15 cockpit for testing.

However, the RRVC must satisfy the additional requirements that have been ongoing in the Virtual Cockpit program. These additional requirements include: photo-realistic cockpit displays, distributed simulation capability, and off-the-shelf computer systems. Table 3-1 contains a more detailed list of the Rapidly Reconfigurable Virtual Cockpit's requirements. The chapter will contain a more detailed discussion of each of the above requirements and a list of design alternatives to satisfy the requirements. Finally, the chapter will contain the design of choice and reason for its selection. Speed, realism, reconfigurability, and maintainability are the decision criteria for the RRVC design.

Given the requirements in Table 3-1, the design for a Rapidly Reconfigurable VC was developed and is shown in Figure 3-1. The design utilizes the CODB architecture, reconfigurable software models, and DIS support. The following paragraphs contain more information on these design choices. The design consists of a CODB that is at the center of the entire RRVC application and is the primary means for data transfer between simulation components. The first simulation component is the VC_Renderer that is a super class of the AFIT_CODB_Renderer. The VC_Renderer is responsible for setting up the Performer environment, managing the top-level Performer geometry tree, and view manipulation. The IO classes on the right side of the figure are responsible for all input into the simulation. Each class handles a different type of input as denoted by their name. The WorldStateManager, at the bottom of the diagram, is responsible for all communication with the DIS network. The Convert_To_Local component is responsible for taking the output from the WorldStateManager (WSMEntityStruct) and converting it into Performer flat-earth coordinates for use by the other VC components. The Airplane component, that takes up the left

Table 3-1. Rapidly Reconfigurable Virtual Cockpit Requirements.

Requirement	Goal
1. Reconfigurable Software Architecture	
1.1. Allow rapid reconfiguration of both aircraft geometry and simulation components (see REQs 2 and 3)	Architecture should allow switching between aircraft in less 1 second.
1.2. Increase flexibility of simulation framework by eliminating ObjectSim's constraints	Allow all Performer functionality to be available to developers by completely removing ObjectSim from VC
1.3. Provide architecture that will support all needed simulation components and be extensible	Utilize container-based approach to storing simulation data
1.4. Support Multiple Aircraft	Configuration supports at least two aircraft
1.4.1. Integrate existing F-15 VC into CODB architecture	CODB F-15 VC works identically to 1995 F-15 VC.
1.4.2. Develop a F-16 VC	F-16 VC with appropriate aircraft model, sensors and weapons (REQ 3).
2. Reconfigurable Cockpit Geometry Models	
2.1 Allow switching between different aircraft cockpits	Switch between F-15 and F-16 cockpits in less than one second
2.2. Create photo-realistic F-16 Instrument Panel	Cockpit instruments same size, shape, and coloring of actual display
2.2.1. Use anti-aliased textures for cockpit text	All text in cockpit is anti-aliased to increase realism
2.2.2. Design textures to be reusable in other aircraft	All text textures can have any foreground or background color and any font size
2.2.3. Create realistic dials and needles for instruments	Entire instrument panel implemented in DWB to allow addition of materials and lighting effects
3. Reconfigurable Simulation Components	
3.1. Develop reconfigurable aircraft aerodynamic model	Aircraft aerodynamic model that can represent many types of aircraft
3.1.1. Model must be able to represent multiple aircraft based on data alone	Model will represent both the F-15 and F-16 aircraft and have data to support additional aircraft
3.1.2. Model must utilize CODB based input and output	All input and output from model is CODB based
3.2. Develop reconfigurable radar model	A simple CODB-based radar model that represents multiple aircrafts' field-of-views.
3.2.1. Radar must be able to change field-of-views while running	Any radar field of view attributes can be changed during execution of the application
3.1.2. Model must utilize CODB based input and output	All input and output from model is CODB based
3.3. Develop reconfigurable weapons controller	Weapons controller will support all current weapons and multiple aircraft
3.3.1. Modify existing weapons controller to support multiple aircraft	A single weapons controller that will support multiple aircraft
3.3.2. Provide CODB container for weapons status data	Weapon status information will be available to all components in CODB
3.3.3. Create new bomb that will be guided to target by a Virtual GPS receiver	Utilize existing bomb model to create an additional bomb type that will use GPS for guidance and hit target
4. Replace ObjectSim Functionality	
4.1. Replace origin-centered viewpoint algorithm	Overcome Performer viewpoint resolution problem
4.2. Structure Performer tree to support weapons model and viewpoint algorithm	Create same top-level tree structure as that in 1995 ObjectSim VC
5. Improved cockpit interface	
5.1. Allow selection of three dimensional panels and instruments	Point and click interface for any type of geometry in a single easy to use class
5.2. Eliminate need for maintaining an active panel for button selection.	Any button can be selected at any time.
5.3. Improve interface with switches and dials	Dials and switches move left / right and slow / fast as desired.
6. Distributed Simulation Interface	
6.1. Send and receive entity state information for aircraft	Communicate Entity State PDU information with DIS using CODB
6.2. Broadcast weapon state information on network	Communicate Fire, Entity State, and Detonate PDUs using CODB.
6.3. Display all network entities to pilot	Network entities appear correctly in Performer scene

side of the figure, is responsible for simulating the entire aircraft. The airplane is a framework for the following models: aerodynamics, weapons, sensors, and instruments. Most of the components in the figure use the CODB for both input and output. Output is of primary importance in this figure to show what information will be available to other simulation components.

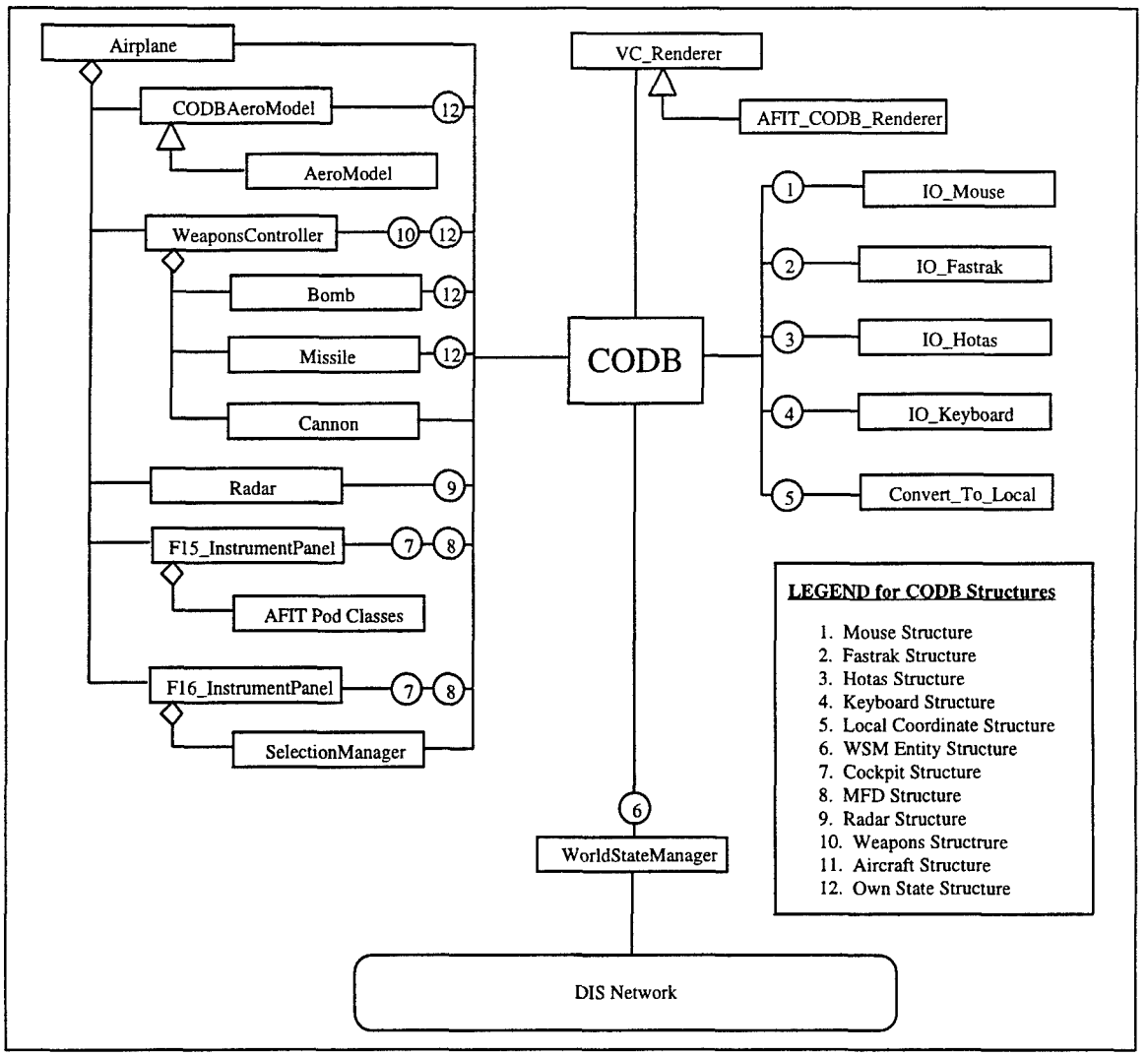

Figure 3-1. Rumbaugh Diagram of Top-Level VC Design.

Note: Diamonds represent aggregation, where the top-level component is made up of the lower level components. Triangles represent inheritance, where the top-level component inherits the behavior of the lower level components. Numbers in circles beside the components represent the primary CODB data structure updated by this component.

3.1. Reconfigurable Computer Architecture

The computer architecture must be able to support the development of multiple aircraft cockpits and allow reconfiguration between them. The architecture must also be able to quickly switch between configurations with minimum impact to performance and be able to support the wide range of functionality that can occur between aircraft. The computer architecture's ability to meet the above requirements will be tested by integrating the current F-15 VC into the architecture and then developing an F-16 VC under the new architecture. Previous VC's utilized ObjectSim, a peer-to-peer architecture. An alternative is the Common Object DataBase (CODB), which uses a centralized architecture to store information. Each of the alternatives has advantages and disadvantages to their use.

The first choice, ObjectSim's primary advantage is that the VC's current implementation uses this architecture. In fact, the current speed and realism are acceptable. ObjectSim also includes a great deal of functionality built into its libraries, including support for I/O devices and distributed simulation. Unfortunately, ObjectSim also has many disadvantages, the primary being ease of use. The library has a considerable learning curve for its many classes and methods. Users must not only understand the ObjectSim classes and methods available, but users must also have a thorough understanding of Performer before they can effectively use ObjectSim. ObjectSim classes are very tightly coupled to Performer and to each other. Tight coupling makes changing the libraries difficult; therefore, adding any needed additional functionality will also be difficult. For instance, the ObjectSim uses the Modifier class for input devices and its purpose is to manipulate the current view of the simulation. If a user wishes to use an input device for a purpose other then manipulating the simulation view they must override certain methods in the class. In addition, ObjectSim provides no support for handling data inside the multiprocessing environment that Performer furnishes to the user. Users expect this because ObjectSim is primarily a simulation toolkit and not data-handling or multi-processing architecture.

The other primary choice, the Common Object DataBase (CODB), is primarily a data-handling architecture. The architecture is relatively new and does not include a large number of classes for simulation development. However, to its advantage, it stresses structured classes to communicate to the

world through a centralized database. While this does not directly reduce the amount of coupling of a simulation, it does reduce the amount of information that a class must maintain about other classes. A central database now contains the data that lead to the coupling. The simulation engineer can abstract away from the classes and methods that will produce needed data. To obtain the desired data the simulation engineer must only concern themselves with the container in the database where the information resides. In addition, the architecture supports double-buffering and multiple processes that are very important in the Virtual Cockpit where four processors are currently required to maintain an acceptable frame rate [DIAZ94]. Double-buffering allows separate application components to read and write from the central database at the same time reducing the waiting time often associated with sharing memory among processes. Disadvantages to the CODB include a limited number of classes that support or fit into the CODB framework and CODB access methods must now be built into all classes needing shared data.

A shared approach could also be taken that would utilize both architectures. The CODB could be used for all shared data access and ObjectSim could be used for setting up network entities. This approach would seem to have many advantages including a large set of simulation classes and good data handling facilities. However, all of the ObjectSim-based classes must be modified to support the CODB. The disadvantages of ObjectSim would also be part of any mixed model approach.

The approach decided upon was to use the Common Object DataBase architecture for the reconfigurable VC. The primary reasons for this decision are the built in data support for multi-processor applications and the abstraction away from classes and to containers (CODB structures). The built in data handling functions use double-buffering and pointers to reduce the time spent waiting of data and the amount of data passed through the system. The abstraction to containers will assist in the reconfigurability of the different aircraft. For instance, if the VC is modeling both a C-5 and a F-15, it may need two separate software models to simulate each aircraft's navigation system. This may mean two separate software models / classes must communicate to the rest of the VC while many other components may be identical (e.g., the radar display on the multi-function display or the radar altimeter instrument). With both radar models communicating through a common container, the radar altimeter instrument will now only need to access the radar's database container instead of determining that class method is needed to obtain

the desired data. In addition, the difficulty in expanding the ObjectSim library was another factor in the choice of the CODB architecture. Expansion difficulties could lead to problems if future VC aircraft types contain characteristics that do not fit into the ObjectSim architecture. For these reasons the Rapidly Reconfigurable Virtual Cockpit uses the CODB architecture for design and development.

3.2. Reconfigurable Cockpit Geometry

A cockpit existed previously for a F-15; however, a F-16 cockpit was required for reconfigurability testing. A photo-realistic F-16 instrument panel was developed to fulfill this requirement. An entire F-16 cockpit is not being developed because the thrust of this research is the rapid reconfiguration between cockpits and not cockpit design. For the instrument panel to be photo-realistic, it must have the same size, shape, and coloring as an actual F-16 cockpit. To allow instrument panel text to appear like that in an actual cockpit, the models use texture maps on simple polygons [DIAZ94]. The text texture maps should be reusable for other aircraft. Reuse of texture maps will save the limited amount of texture memory that is available (Note: texture memory is a portion of system memory allocated to textures, if the amount allocated is exceeded the performance of the application is degraded as texture memory must be swapped out when a new (unloaded) texture comes into view). To increase realism, all dials and needles were developed in DWB to allow effects not easily achievable with Silicon Graphics' Graphics Library (GL) calls. Finally, the application must be able to switch between virtual cockpits quickly. Fortunately, Performer contains a construct that allows the switching to take place very quickly.

An immediate design issue in the development of the F-16 models was how textures would be created and used to create photo-realistic displays. Diaz had shown that texture maps of the words on the instrument panel could provide anti-aliased and photo-realistic cockpit instruments [DIAZ94]. However, the textures that were developed for the F-15 VC are not usable for the F-16 VC because of both content and coloring. Diaz had created the textures by using a paint program to place white text on a background the same color as the F-15 instrument panel. One choice for texture development was to develop a whole new set of textures for the VC using the same method as Diaz had. The other choice was to develop textures that are suitable for the VC and that other aircraft cockpits can use. The requirement called for

reusable texture that can be employed on many different cockpits. The textures must support any foreground or background color to allow them to support reuse. The textures are created by using a white text color that when placed on a colored polygon will allow the text to take on the color of the texture-mapped polygon. The non-text portion of the texture will be totally transparent; allowing it to be the color of the background polygon (the polygon behind the texture-mapped one). By using the correct coloring scheme and utilizing the alpha component of textures, text-based textures are created that can be used in many different Virtual Cockpits.

Another design point in the development of the F-16 instrument panel was the degree to which the displays would be realistic. In the F-15 VC, all of the components of the cockpit that were being updated by the aircraft model were drawn utilizing Silicon Graphics' Graphics Library (GL) calls. Using GL allowed the drawing of needles and numbers where needed in the cockpit. However, the numbers were a vector font and were not anti-aliased, which decreased the realism associated with anti-aliased fonts. In addition, the GL needles did not have the same materials as the other components of the cockpit and therefore were not affected by changes in lighting or shadows. Dials were also implemented using a GL vector font; numbers on the dials were updated based on the instrument's current setting. However, the dials did not spin, which prevented the pilot from noticing the motion or the rate at which the value was changing unless they were staring directly at the values as they changed. Therefore, spinning dials were implemented in DWB to provide a realism not easily be duplicated using the available GL calls. Finally, the cockpit utilizes many DWB models to increase the realism of the F-16 VC.

3.3. Reconfigurable Simulation Components

The Virtual Cockpit utilized reconfigurable simulation components to aid in the development of different Virtual Cockpits. Reconfigurable simulation components are models that can represent different configurations or types of the same model. For instance, a single reconfigurable aircraft model can represent several different aircraft. The type of component a reconfigurable component represents is usually set in one of two ways, at creation or by changing the parameters used to drive the model. Reconfigurable models developed for this research, include an aerodynamics model, a weapons controller,

and a radar. Two separate instrument panel components represent the F-15 and F-16 cockpits, because of the large differences in appearance and functionality between the two cockpits. The current instrument panel in use depends upon the RRVC's current aircraft type. The Airplane class contains and updates all models.

The use of a "C" language aircraft model from Wright Laboratory's Flight Simulation Facility (WL/FIGD) that was highly parameterized greatly helped the development of a reconfigurable aircraft model. The model utilizes data files to determine the type of aircraft it will represent in a simulation exercise. The original model initializes as a single type of aircraft and is flown as that aircraft for an entire simulation. To use the model to rapidly reconfigure from one type of aircraft to another the model must under go the following transformations.

1. Change model to an object-oriented C++ model to support the declaration of multiple aircraft.
2. Change the model to allow setting the aircraft type at creation time.
3. Change the model to utilize the Graphics Laboratory's combination stick, throttle, and rudder input device.
4. Change the model to allow user to reset the position of the aircraft to any orientation or position.
5. Change input and output for the model to support the CODB architecture.
6. Change the model to include support for the Performer coordinate system.

After making these changes, multiple aircraft types can be created using this single model. Expansion of the ResetPosition function to allow all orientation information will allow an aircraft model to start out in another aircraft type's orientation and position. Other reconfigurable models developed include a radar and weapons controller.

A simple radar model design provides a basic radar functionality. The VC's current radar uses a simple viewing frustum and has two modes, one for air entities and one for ground entities. If an entity is in the frustum then it is seen by the "radar" and its coordinates are transformed by scaling, rotating, and translating them to the correct position on the radar display. Disadvantages of the current design include hard-coded values for radar range, radar field-of-view, and not maintaining azimuth and elevation information for the radar's tracks (Note: typical radars return track azimuth, elevation, and range). This a disadvantage because other aircraft components such as displays need azimuth and range information. Hard-coding the values also prevents the application from being able to change the values to alter the radar's search volume, that is a typical pilot operation in the F-15 and F-16. A navigation class contains

and hides the radar model. This not only makes the radar model difficult to find but also difficult to change. The new design will create a separate radar class that matches the design in Figure 3-2. The design encapsulates the radar in its own class and provides methods for setting radar FOV and range. The radar model uses a frustum in the same way as the current model. In addition, the F-16 VC's radar display will use an azimuth versus range display, as opposed to the current situational display employed in the F-15 VC. Figure 3-2 also shows the CODB container that stores radar output. The design provides a reasonable radar interface and functionality until a more realistic model can be developed.

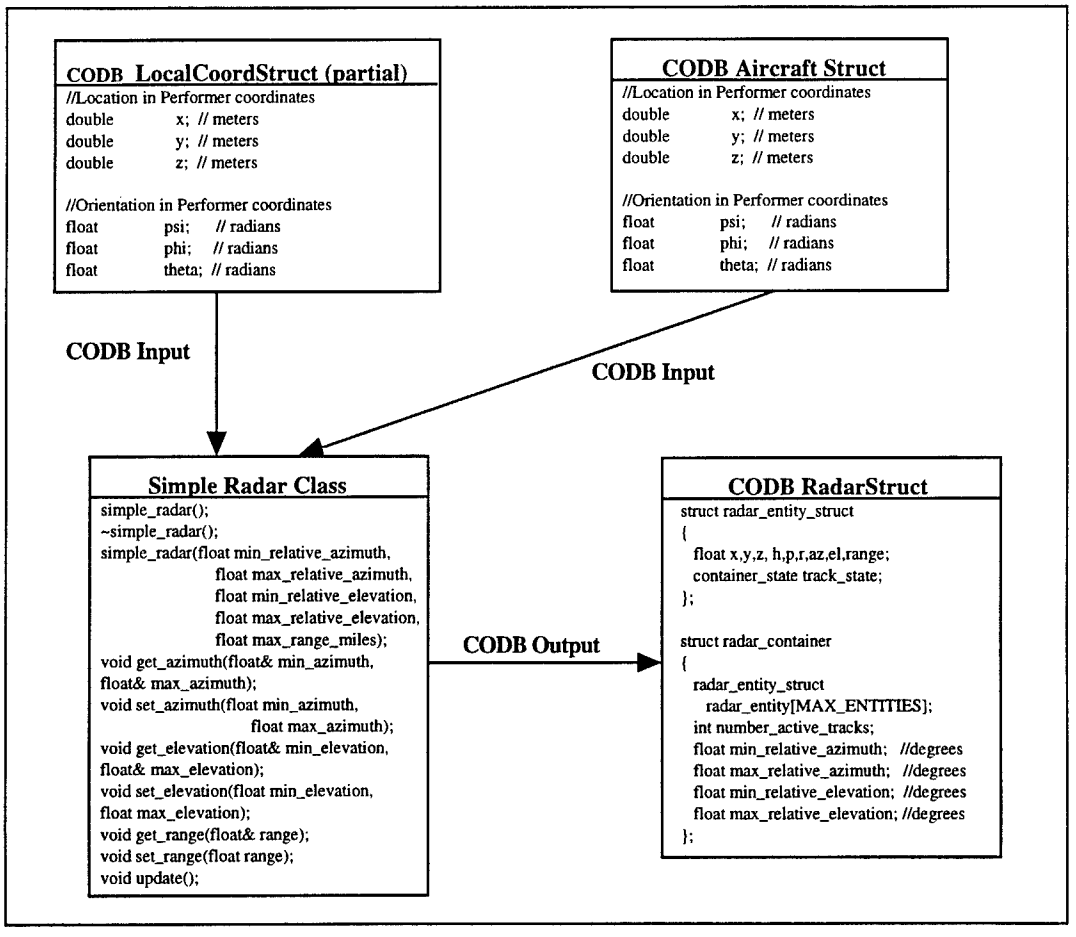

Figure 3-2. Radar Class with CODB Input and Output Structures

The weapons controller class existed previously in the F-15 VC; however, due to some hard-coded values, it cannot be used for separate aircraft. The model was changed to allow multiple weapon controllers to be created, each with separate weapon loads. In addition, maximum weapon type limits were increased

to allow the weapons controller to be used for the F-16 aircraft. The model will be changed to interface with the new CODB interface and will output weapons information to the CODB WeaponsStruct container. A final change to the weapons controller is to allow it to interface to the new DIS manager for broadcasting weapons state information across the network. The Distributed Simulation Interface section of this chapter discusses these changes in more detail.

The Weapon Controller's bomb class is already a functional class that enables it to simulate several different types of bombs. A previous thesis student developed the class and it provides several types of simple bomb models, including both guided and dumb bombs [GERH93]. To create a GPS-guided bomb (Requirement 3.3) the experimental bomb type, WXG will be changed to represent a GPS-guided bomb. The bomb will use a Virtual GPS Receiver developed by Captain Gary Williams to determine its GPS Position and use that position for guidance [WILL96]. The bomb model keeps track of both its true position and its GPS position. The bomb updates its GPS position by calling the Virtual GPS Receiver's method get_gps_pos and passing it the bomb's true position and the current simulation time. Only a small portion of the bomb model changes when incorporating the modifications stated above.

The F-15 and F-16 VC's will have two separate instrument panel components because of the great difference in functionality between the two cockpits. The instrument panel components are responsible for updating the displays and controls of the cockpit. Two approaches could have been taken in the development of an instrument panel component. First, a single reconfigurable software class for all the different aircraft types that can be represented by the RRVC. The second approach is to use a separate model for each aircraft type. The second approach was chosen for several reasons, primarily because of the tight coupling between this class and the actual geometry files used to model the instruments. This class is responsible for manipulating the Performer trees that make up the instrument panel based on the current state of the aircraft. Because each individual aircraft type's cockpit is very different from another, the decision was made to have separate instrument panel classes for each aircraft type. This will encapsulate the aircraft type's differences in their own individual classes and eliminate the complexity associated with a single mammoth instrument class for all aircraft types. Keeping the classes separate also allows most of the current F-15 VC instrument panel code to remain unchanged, except for CODB additions. The F-16 VC

can then use the new Selection Manager discussed in section 3.5, without being tied to the framework of the current architecture.

3.4. Replacing ObjectSim Functionality

The primary replacement of ObjectSim functionality involves the updating of simulation entities and the integration of terrain files into simulation. The design approach was to utilize a Performer tree similar to the 1995 VC, that maintains the VC, the DIS Players, and the Terrain all as separate Performer trees, see Figure 3-3. The VC_Renderer class encapsulates all top-level Performer tree activity. A second ObjectSim replacement also was placed in the VC_Renderer class and involves moving the terrain and DIS players relative to the VC. The code from the ObjectSim pfmr_renderer class is used to duplicate this functionality. Movement of the world around the VC is necessary because of the way Performer updates the viewpoint in the simulation. Performer uses a pfVec3 to update the viewpoint that is an array of three floats for x, y, and z. As these numbers increase in size, floating points lose resolution and the viewpoint begins jumping around, resulting in movement that does not appear smooth to the user. The VC_Renderer uses an algorithm from ObjectSim's View class to update the view, shown below.

```
-- Algorithm to move the viewpoint of an aircraft to the origin and to move other entities
-- and terrain relative to the aircraft. The algorithm also considers an offset.
-- Set Position of OriginCoord to origin (x = y = z = 0.0)
        PFSET_VEC3(OriginCoord.xyz, 0.0f, 0.0f, 0.0f);
-- Set Orientation of OriginCoord to Aircraft's Orientation
        PFCOPY_VEC3(OriginCoord.hpr, Aircraft->Coords->hpr);
-- Translate and Rotate Aircraft's Geometry by OriginCoord
        pfDCSCoord(My_Model->RotDCS, &OriginCoord);
-- Set OriginCoord to Aircraft's orientation plus any offset
        PFADD_VEC3(OriginCoord.hpr, aircraft->Coords->hpr, offset->base_rot);
-- Set Position of OriginCoord to origin (x = y = z = 0.0)
        PFSET_VEC3(OriginCoord.xyz, 0.0f, 0.0f, 0.0f);
-- Make a coordinate transformation matrix from OriginCoord
        pfMakeCoordMat(viewmat, &OriginCoord);
--Transform the Offset by coordinate transformation matrix
        pfXformPt3(Result, (*attached)->base_offst, viewmat);
-- Set View parameters
        pfChanView(chan,  Result, OriginCoord.hpr);
--Negate the resulting transform (opposite direction of aircraft)
        PFNEGATE_VEC3(Result, (*attached)->Coords->xyz);
-- Move the players in the opposite direction as aircraft
        pfDCSTrans(playertrans,  Result[PF_X], Result[PF_Y], Result[PF_Z]);
-- Move the terrain in the opposite direction as aircraft
        pfDCSTrans(terraintrans, Result[PF_X], Result[PF_Y], Result[PF_Z]);
```

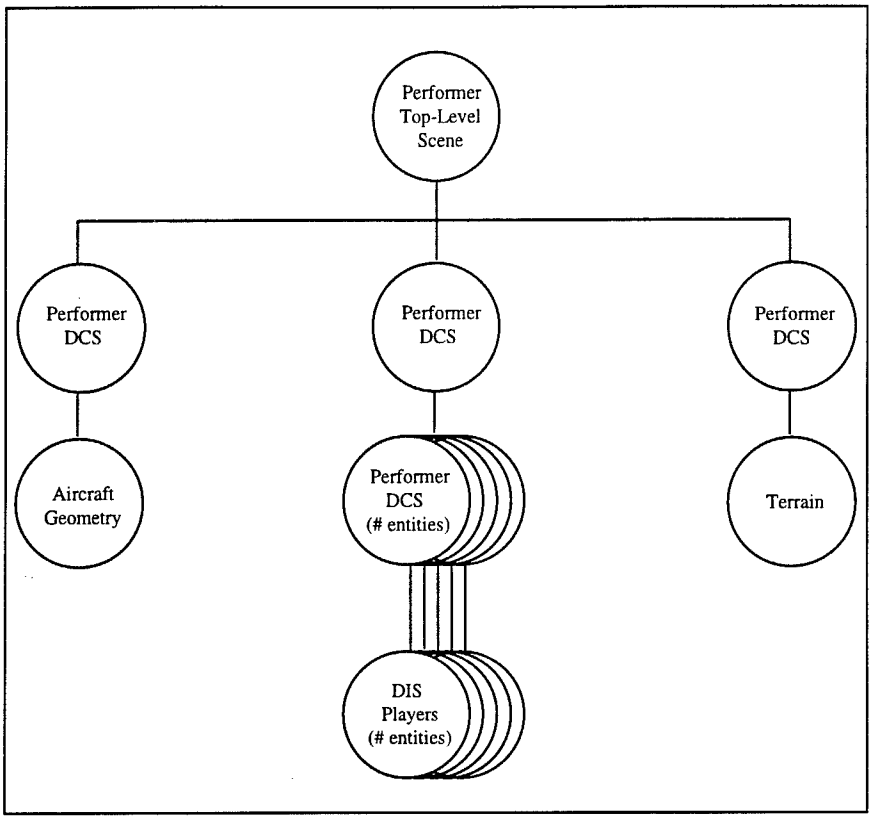

Figure 3-3. 1995 VC and ObjectSim Performer Tree

3.5. Replacing AFIT Pod Interface

The requirement for natural interaction with the aircraft cockpit has not changed from the inception of the VC research. The current design and implementation refers to the AFIT Pod. This design utilizes a series of classes that implement panels, sub-panels, buttons, and a mouse used for selection [KEST94]. For an application to interact with a single button using the AFIT Pod, the developer must create a panel and place a sub-panel on that panel and place a button on that sub-panel and then register the panel with a mouse previously created. To select a button, the application creates a line segment based on the position of the mouse and the position of the panel. The application passes the line segment to each of the classes above to test for intersection of the button and the line segment. This process expands if the user wishes to define their own button shape, such as a switch or dial. A further limitation is that a a panel may only be on

a single plane. For a system to have multiple control surfaces they must create multiple panels. Unfortunately, a user cannot select a button on any panel; but, can only select a button on the current active panel. Therefore, the user must switch between panels in order to select a button on an individual panel. As can be seen, this process is very complicated. The developer must create a large framework for a single button and the amount of automation gained from utilizing this framework is relatively small. The AFIT-Pod framework also limits the user to two-dimensional selection that is not acceptable in a three-dimensional virtual environment, such as the RRVC. The two-dimensionality of the AFIT Pod prevents the user from selecting on other surfaces unless the developer creates an additional AFIT Pod Panel. The user of the system must then switch between panels before they can make a selection, meaning only certain things can be selected at certain times. While the AFIT Pod provides some basic functionality, its disadvantages greatly outweigh its advantages to the RRVC research.

A new design developed in conjunction with Captain Brian Garcia will greatly simplify the approach and replace all the above classes with a single class [GARC96]. The new Selection Manager class will make use of optimized picking functions built into Performer and eliminate the complexity currently associated with the process. A Performer function is available that performs an intersection through the visible Performer geometry based on the position of the mouse. If any Performer geometry was "intersected" by the mouse then the function returns the path through the Performer tree leading to that geometry. By naming the nodes to be selected (such as switches and dials) with a certain prefix and integer, the new Selection Manager notifies the application when a user picks any of the selectable geometry. By using geometry for selection, the new Selection Manager can pick any type of shape of object anywhere in the scene. The capabilities of the new Selection Manager greatly ease the development process and create additional methods for cockpit interaction (discussed in detail in Chapter 4).

3.6. Distributed Simulation Interface

The new Distributed Simulation interface will continue to use the Distributed Interactive Simulation (DIS) protocol. Originally, the VC was to have transitioned to HLA during this research period; however, the lack of a completed HLA architecture prevented this from happening. The VC will use the

newest version of the World State Manger for its DIS interface, World State Manager 3.0. The World State Manager was created by Sheasby and provides an application with an interface to the DIS environment[SHEA92]. Sheasby has continued to update the World State Manager libraries and they now support the CODB architecture. The VC will interface to the DIS architecture through the supplied CODB DIS structures. The structures include a structure to broadcast local entity state information (OwnStateStruct) and a structure to store network entity state information (WSMEntityStruct). An additional component of this interface is a function that converts DIS entities' positional information from earth-centered coordinates to a flat world representation used by the VC. The function will store a structure identical to the WSMEntityStruct in the CODB and refer to it as the LocalCoordStruct. The creation of an additional storage location will allow any component of the simulation to obtain earth centered coordinates and flat-earth coordinates for any entity in the simulation. Currently the Airplane, VC_Renderer, InstrumentPanel, Radar, WeaponsController, Bomb, Missile, and Cannon classes all require flat earth coordinates. The design will eliminate the need for multiple simulation components to calculate their own flat-earth positions from the DIS coordinates.

3.7. Conclusion

The design of the RRVC meets all of the requirements discussed in Table 3-1. The design focuses on the CODB and simulation components that communicate between each other using the CODB. The design removes ObjectSim components and replaces the architecture with a container-based approach. The computer architecture incorporates the functionality of the F-15 VC and provides interfaces for the F-16 VC. The architecture supports not only reconfigurable models; but, also aircraft unique models for improved realism. The F-16 VC's cockpit design focuses on photo-realism and on creating reusable components. Reconfigurable simulation components are used to allow multiple aircraft types to be simulated and eases future modeling efforts. The RRVC removes the AFIT Pod interface and replaces it with a Selection Manager that is easier to use and implement. Finally, the World State Manager provides DIS support for the VC using the CODB for data transfer. This design allows creation of a reconfigurable

cockpit that allows a pilot to switch between a F-15 and F-16 and take part in a distributed interactive simulation.

4. Implementation

This chapter discusses the implementation of the requirements and design discussed in Chapter 3. An incremental build approach is taken in the implementation of the design, with each design component of the VC representing a separate phase of the development. The design components that make up this research are the reconfigurable software architecture, reconfigurable cockpit geometry models, reconfigurable simulation components, replacing ObjectSim functionality, replacing the AFIT Pod Interface, and interfacing to DIS. Each design component builds upon a working baseline, with the initial baseline represented by a CODB Demonstration application. Each of the design components is implemented independently of each other and builds upon the progress of the previous component's implementation. This approach allows incremental testing of the application after each phase of the implementation. During incremental testing, each new component of the RRVC is tested, as well as its interaction with all previously implemented components. The only variation from this approach was the replacement of ObjectSim functionality, which was addressed as needed. The design components were implemented in the same order as discussed in Chapter 3: reconfigurable software architecture, cockpit geometry, simulation components, replacement of ObjectSim functionality, replacement of the AFIT Pod interface, and finally the distributed simulation interface.

4.1. Reconfigurable Software Architecture

The first task in the research project was to move the 1995 VC into the CODB architecture. Primary concerns in this task are the removal of all ObjectSim architecture components and insuring that no functionality was lost in the conversion. Instead of removing ObjectSim references one-by-one and converting them to the CODB, the approach taken was to remove all the VC and rebuild it from the ground up using the CODB architecture. Breaking down the development into manageable components kept the process from becoming overwhelming. Each component's structure and functionality could be examined on its own -- allowing better component-level understanding. The CODB Demonstration application, developed as an Advanced Computer Graphics Course (CSCE 682) project, was used as a working baseline

for the Rapidly Reconfigurable VC (RRVC). The CODB demonstration project included a simple Performer renderer, aircraft model, and I/O devices all integrated into the CODB architecture. The CODB Demonstration application provided both a sample of the CODB architecture and an application that could be built upon to create the RRVC.

The first step in integrating the 1995 VC into the CODB application involved breaking apart the VC into components. Originally it was thought that each class could be taken out of the 1995 VC and turned into a CODB class; however, this was not the case. The design of the 1995 VC involved components that were made up of several highly coupled classes that communicated via a shared memory structure. These components were identified as the cockpit, weapons, head-up display (HUD), aircraft model, and multi-function display (MFD) / inertial navigation system (INS) / radar. The primary shared memory structures used extensively throughout the VC were a MFD/INS/Radar structure and a Cockpit structure. The cockpit component of the VC included classes for the aircraft's front panel and side consoles. These classes were built upon the AFIT Pod classes (panel_type, sub_panel_type, button_type, button_type2, and mouse) for button selection purposes. Additional Cockpit component classes existed to keep track of the cockpits current status using the Cockpit structure. The weapons component included a weapons controller along with classes for a cannon, bombs, and missiles. The weapons component utilizes part of the MFD/INS/Radar structure to keep track of targeting information. The other components of the system were three tightly coupled classes (MFD, INS, Cursor) that controlled the radar, kept track of navigation points, and displayed information on the MFD's. The MFD, INS, and Cursor classes communicated via a shared memory structure that contained all the information of interest to the classes. The HUD used information about aircraft position and orientation along with data in the MFD/INS/Radar structure for targeting information. A class that did not utilize shared memory structures to any great degree is the aircraft_model. The aircraft model used only the input of the throttle and stick to control it and used methods to communicate its status.

Once the 1995 VC was broken down into its components, the cockpit component was chosen as the first component to integrate into the RRVC. The cockpit component did not utilize any ObjectSim

classes and therefore was relatively easy to integrate into the CODB Demonstration application. However, the effort required more than simply compiling the class and ensuring the geometry was in the correct place in the cockpit. A CODB container was created to replace the shared data structure. The replacement allows each class of the component to communicate through the CODB. Access calls to the CODB replace all references to the shared data structure. The cockpit component while fairly complicated was well encapsulated and provided a good first attempt at transitioning a component into the CODB architecture.

Integrating the cockpit component further highlighted several advantages of the CODB architecture and disadvantages of the ObjectSim design. The primary advantage of the CODB is the double-buffering it provides. Double-buffering allows two processes to read and write to the same data structure at the same time and eliminated the need for different processes to share one set of data. In the 1995 VC, the process responsible for drawing the geometry was sharing the data with the process responsible for updating the data. Semaphores were used to protect the data and could cause the processes to wait on each other, eliminating some of the advantages associated with the multi-processing capabilities built into Performer. Another advantage of the CODB is that it maintains shared state information in shared memory and allows access by any process. This eliminates the need to keep two copies of the data, one in local memory and one in shared memory, as was the case with the 1995 VC. A disadvantage of the 1995 VC design discovered during the conversion to CODB was the how complicated the current AFIT Pod interface made the F-15 VC application. The F-16 component of the RRVC will utilize a new selection interface that will be discussed later in this chapter.

The second group of components to be integrated into the VC were the HUD component and the MFD/INS/Radar component of the F-15 VC. The HUD was changed to read the aircraft state from the CODB. The HUD utilized the MFD/INS/Radar shared memory structure for targeting information. This structure was left in place to ease integration of the HUD and replaced with a CODB structure after the MFD/INS/Radar component was integrated into the CODB. The next step was to integrate the MFD, INS, and Cursor classes into the CODB. The process was straight forward, in that all shared memory accesses were replaced with CODB accesses. However, this component was created in the same manner as the

cockpit component, all data that had to be accessed in separate processes was stored in both local and shared memory. The classes were simplified by eliminating all variables / data that were duplicated. Eliminating the duplicated variables made the class easier to understand and eliminated the added effort required to ensure both sets of variables agreed with each other.

The final component integrated into the simulation was the weapon component. The weapon models were eliminated from the VC in 1994 and had not been used since. Integration of this component not only involved conversion to CODB, but also compliance with the current VC code. The CODB integration was straight forward because of limited shared memory use and therefore only a few CODB accesses. However, all the weapons software had to be inspected for data validity, ensuring that all data being used by the weapons was available and valid. For testing purposes, the weapons were exercised and examined for correct execution. The models work as they were designed to; however, the models are based on simplifying assumptions that sometimes results in unrealistic behaviors. Additional changes to the weapons models were needed to make the weapons controller support multiple aircraft, that is covered in the Reconfigurable Software Models section of this chapter. After all components were moved to the CODB architecture an Airplane class was created to provide a framework for all of the components, both individual aircraft components and reconfigurable components. Figure 4-1 provides a view of the types of instance variables and methods included in the Airplane class. The Figure 4-1 contains all instance variables for both the F-15 and F-16 aircraft. The beginning of the columns contain cockpit variables and geometry selection classes. The middle portion of the columns contain components that are shared: Round_Earth_Utils (conversion from DIS coordinates to flat earth coordinates), DIS_Manager (responsible for all network interfaces), VC_Renderer (maintains all geometry variables for simulation), and VC_Switch_Node (the Performer node used to switch between F-15 and F-16 geometries). The bottom portion of the left column contains the methods which are used to operate the class -- their functionality is described in Table 4-1.

Table 4-1. Airplane Class Methods and Functionality.

Airplane	Constructor - creates instance of class
Initialize	Initializes all instance variables in class
Update	Performs one frame of simulation
UpdateDIS	Updates Aircraft's DIS Position through World State Manager
Create_Performer_Subtree	Reads in aircraft's geometry files and creates a Performer tree
Get_Performer_Root	Returns top node of Airplane subtree
Get_DIS_Root	Returns the portion of the tree used for DIS components
ResetPosition	Resets position of aircraft to desired position
Get_Performer_Position	Returns Performer Position and Orientation
Get_Performer_Velocity	Returns Performer Velocities for Position and Orientation
Reload_Weapons	Convenience function to reload weapons while running
Reset_Aircraft_Type	Change type of aircraft being flown
Draw	Portion of class which must be updated in draw thread

The Airplane is intended to be a multi-processing Performer component and is created with an Update method, to be called in application process, and a Draw method, to be called in draw process. The performance of the default cull provided by Performer eliminates the need for a custom cull method. As can be seen in Figure 4-1 the components are split between F-15 components and F-16 components. For reconfigurable components such as the aircraft model only a single type of class reference is needed and is called regardless of the type of aircraft (F-15 or F-16) being flown (Note: two CODBAeroModels are created, one for F-15 and one for F-16, that is then accessed via the variable vc_aero_model). However, for non-reconfigurable classes such as the instrument panel, a case statement provides a way of only calling the class associated with the current type of VC aircraft being flown. Non-reconfigurable models allow aircraft specific models to be integrated into the architecture without trying to make them be parameterized for every type of aircraft. The most important method in the RRVC is the Reset_Aircraft_Type that reconfigures the entire class for another aircraft. Reset_Aircraft_Type takes the type of aircraft as an argument and then changes the VC's primary state variable (Current_VC_Type). Changing Current_VC_Type causes the class to change all the simulation models and switch the geometry to the correct aircraft/cockpit combination. Current_VC_Type is used by each of the class methods to determine what aircraft the model is representing and, therefore, which classes should be updated.

//Types enum vc_type {F15_VC, F16_VC}	
//F-15 Components Left_Console Right_Console Instruments Left_Panel Right_Panel Instrument_Panel Mouse MFD	//F-16 Components F16_Instruments Selection_Manager
INS	Simple_Radar
HUD	F16_HUD
F15_Weapons	F16_Weapons
F15_Root_Node	F16_Root_Node
F15_AeroModel	F16_AeroModel
//Shared Components Round_Earth_Utils DIS_Manager VC_Renderer VC_Switch_Node	//Shared Components Round_Earth_Utils DIS_Manager VC_Renderer VC_Switch_Node

//Methods
Airplane(Round_Earth_Utils*, DIS_Mgr*);
void Initialize();
void Update();
void UpdateDIS();
pfGroup* Create_Performer_Subtree();
pfGroup* Get_Performer_Root();
pfGroup* Get_DIS_Root();
void ResetPosition(x,y,z,mach,heading);
pfCoord Get_Performer_Position();
pfCoord Get_Performer_Velocity();
void Reload_Weapons();
void Reset_Aircraft_Type(vc_type);
static int Draw(pfTraverser*,void*);	

Figure 4-1. RRVC Airplane Class.

4.2. Reconfigurable Cockpit Geometry Models

This section covers the development of the geometry models that were developed to implement the F-16 instrument panel. The F-16 front-instrument panel has to be photo-realistic and use in a small enough number of polygons to be rendered in real-time in the application. Textures are used to reduce the number of polygons needed to create realistic models. Therefore, texture creation proved to be an important part of the F-16 cockpit development, including finding an application to create them. To cover both of the two primary cockpit design issues, creation of textures and photo-realism, the development of a single F-16 instrument, the altimeter, will be discussed in detail. Development of all other displays is simply an extension of the lessons learned during the implementation of the altimeter.

The first step in developing a cockpit, prior to developing instrument models or textures, is finding a realistic copy of a cockpit to model. This paragraph provides a background of how cockpit dimensions and layout were obtained for the F-16 and may be useful for others going through a similar undertaking. A useful source of information was Randy Olsen from the Aeronautical Systems Center's Unit Training Device Program Branch (ASC/YWPD), who provided engineering drawings of an F-16 WTT simulator cockpit. These engineering drawings provided dimensions of both the F-16's front instrument panel and the cockpit's shell. A Critical Design Review document for the F-16 Unit Training Device program, in the same branch, provided drawings of all the instruments in the cockpit along with switch and dial positions to 0.001 inches. Combining these two resources allowed accurate positioning of dials and switches in the cockpit. The next step was to position the instruments in the correct position on the instrument panel, this information was obtained by using drawing and diagrams provided in the F-16 Flight Manual. F-16 Flight Manuals are a tightly guarded resource, not from a security standpoint, because of scarcity. The F-16 System Program Office (SPO) has all its unclassified manuals on CD-ROM and provided limited access to the CD-ROM for this research. Finally, an actual F-16 cockpit, from a stressed airframe, was discovered in the Wright Laboratory Simulation Branch (WL/FIGD) that provided another way to measure and compare the DWB models for accuracy. These sources provided invaluable information that was not available on previous VC efforts; but, is necessary for realistic cockpit modeling and pilot training. If a pilot is bothered by an inaccuracy in cockpit modeling then their concentration will stray from the training to be accomplished and may instead focus on inadequacies in the cockpit model.

The F-16 cockpit utilizes textures for a majority of the text displayed in the cockpit (see Figure 4-2). Diaz used textures in the 1994 VC to provide realistic looking text without the cost associated with using GL or Performer to actually create letters using geometry, see Figure 4-3 [DIAZ94]. The F-16 uses the same approach, but uses a texture's alpha component to create a texture that can be used with different foreground and background color combinations. The ability to vary foreground and background colors is important for reuse. Because while cockpits often have common text string, they often have the text in different colors or shades of colors. Creation of textures involved experimentation of several

different tools before settling on GIMP (General Image Manipulation Program). The tool had to be able to create textures with text in differing sizes and fonts, save in DWB texture format (RGB or RGBA), and be able to set a texture's alpha channel. DWB allows creation and manipulation of textures, but does not support entering text in textures. Super Paint on the Macintosh allows the creation of text-based images; but does not allow a file to be saved in the proper format and involves moving the texture between different platforms. Image Magic, a free UNIX program, provides text insertion and a multitude of file formats; but, no easy method to set an alpha channel. GIMP, a free UNIX program, provides all the needed abilities and was chosen as the tool to create textures even though the beta release occasionally crashes.

Once the right tool was chosen the process of creating the textures began. Multiple foreground colors is achieved by simply choosing in DWB how the texture will be applied, decal or modulate. If a texture is placed as a decal on a polygon, the texture will maintain its same coloration. However, by modulating the texture on the polygon the texture colors are blended with the color of the polygon. By making the F-16 textures have white letters, the letters could be made any color by simply changing the textured polygon's color. Allowing multiple background colors is made possible by using the alpha channel to make everything that was not a letter in the texture completely transparent by setting its alpha component to 1 (black). Placing a texture on a yellow polygon will cause the text in the texture to become yellow and the alpha component of the texture would make the rest of the polygon transparent. Placing this polygon on top of any other colored polygon would make the background color the same as that polygon. The polygon must be separated by some distance or their coplanarity will result in a shimmering effect in the Performer simulation. This minimum distance was found to be 0.01 inches at the Virtual Cockpit view distances. The step-by-step process of creating a texture for the altimeter is shown in the following figures:

Figure 4-2. Placing the numbers for the altimeter into an image.

Figure 4-3. Inverting the image to make letters white (0) and background black (1).

Figure 4-4. Creating a second image that is entirely white (0).

Figure 4-5. Saving the second image as an RGBA file using the first image as the alpha channel.

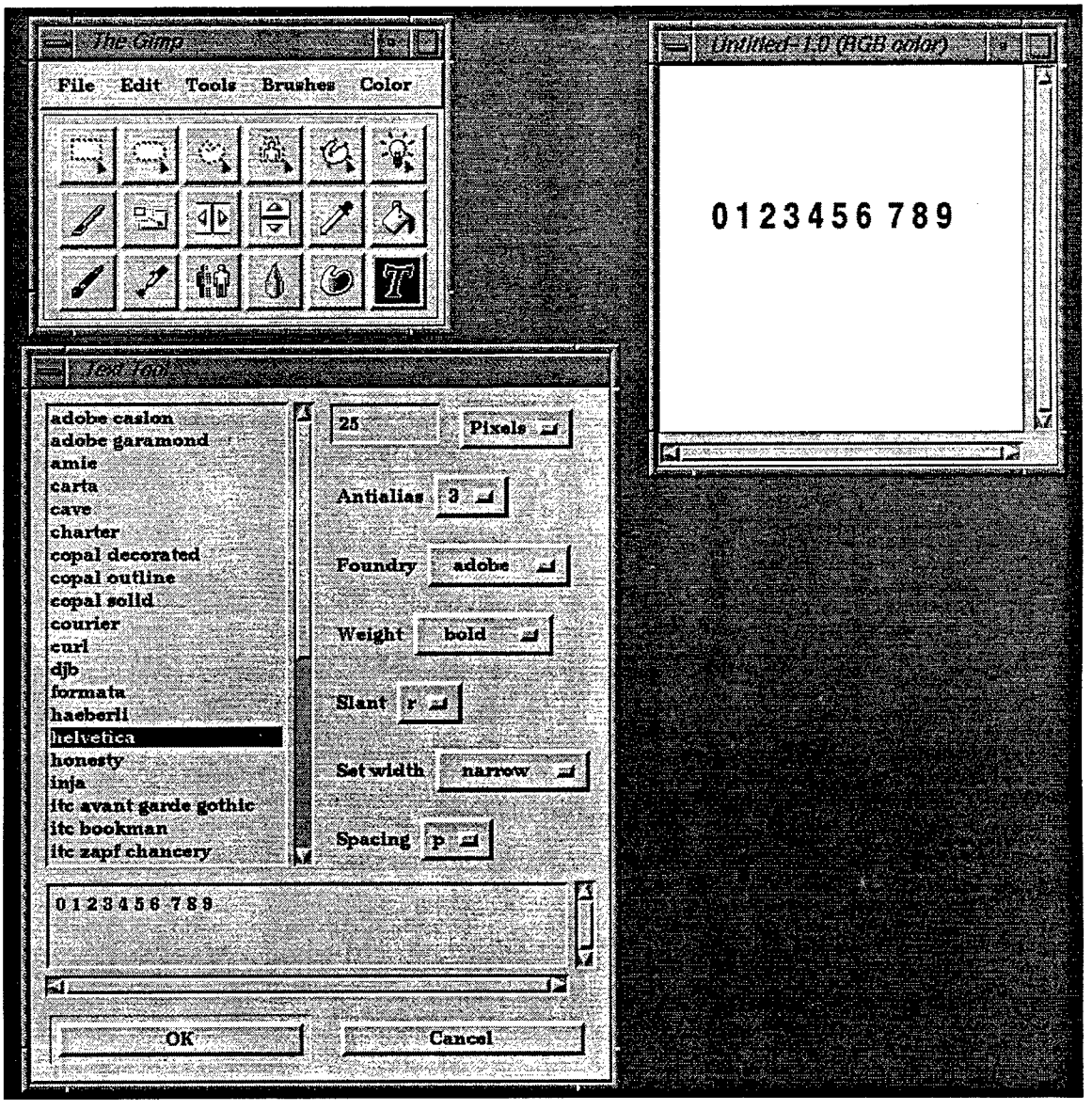

Figure 4-2. Placing The Numbers For The Altimeter Into An Image.

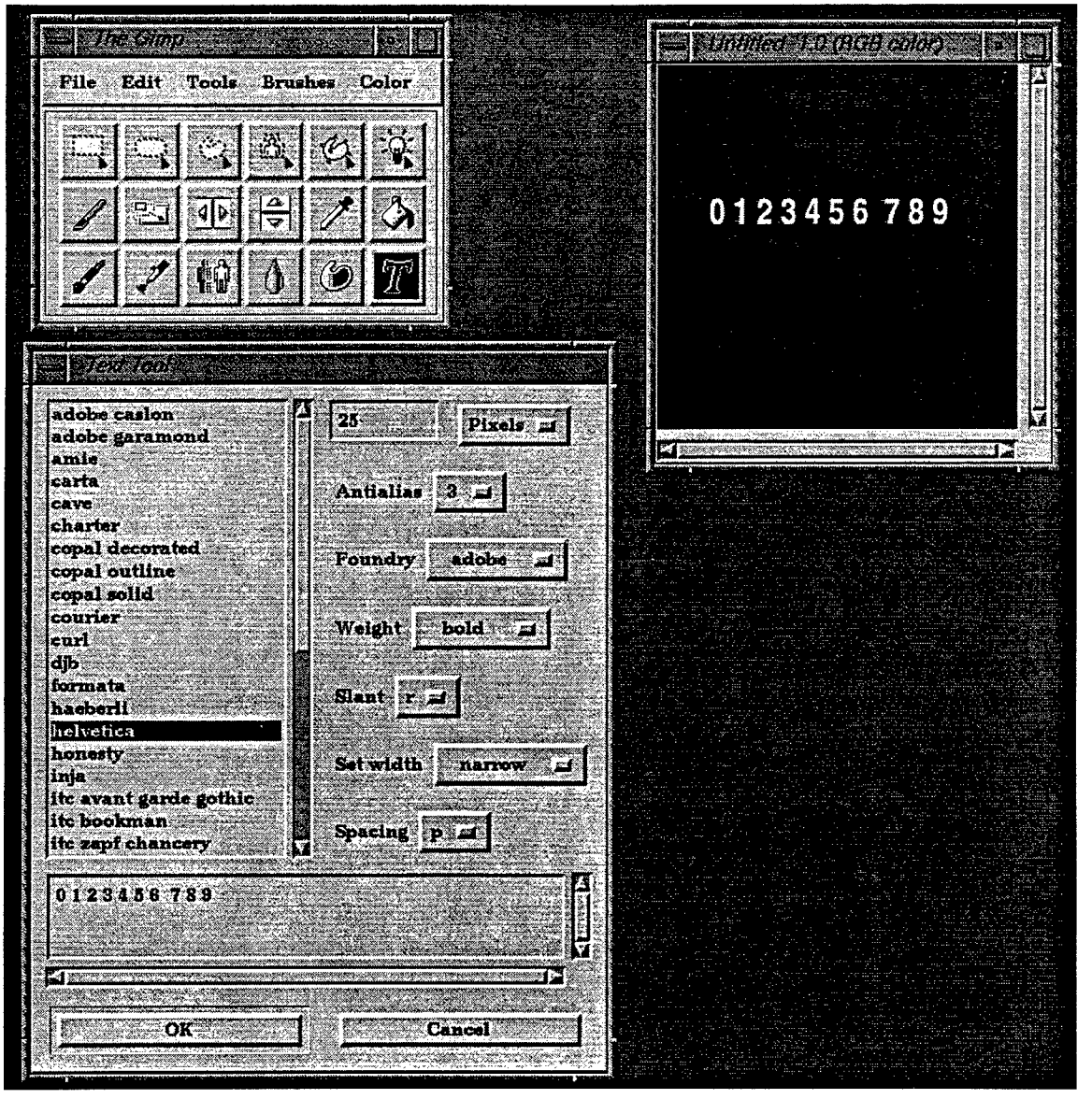

Figure 4-3. Inverting The Image To Make Letters White (0) And Background Black (1).

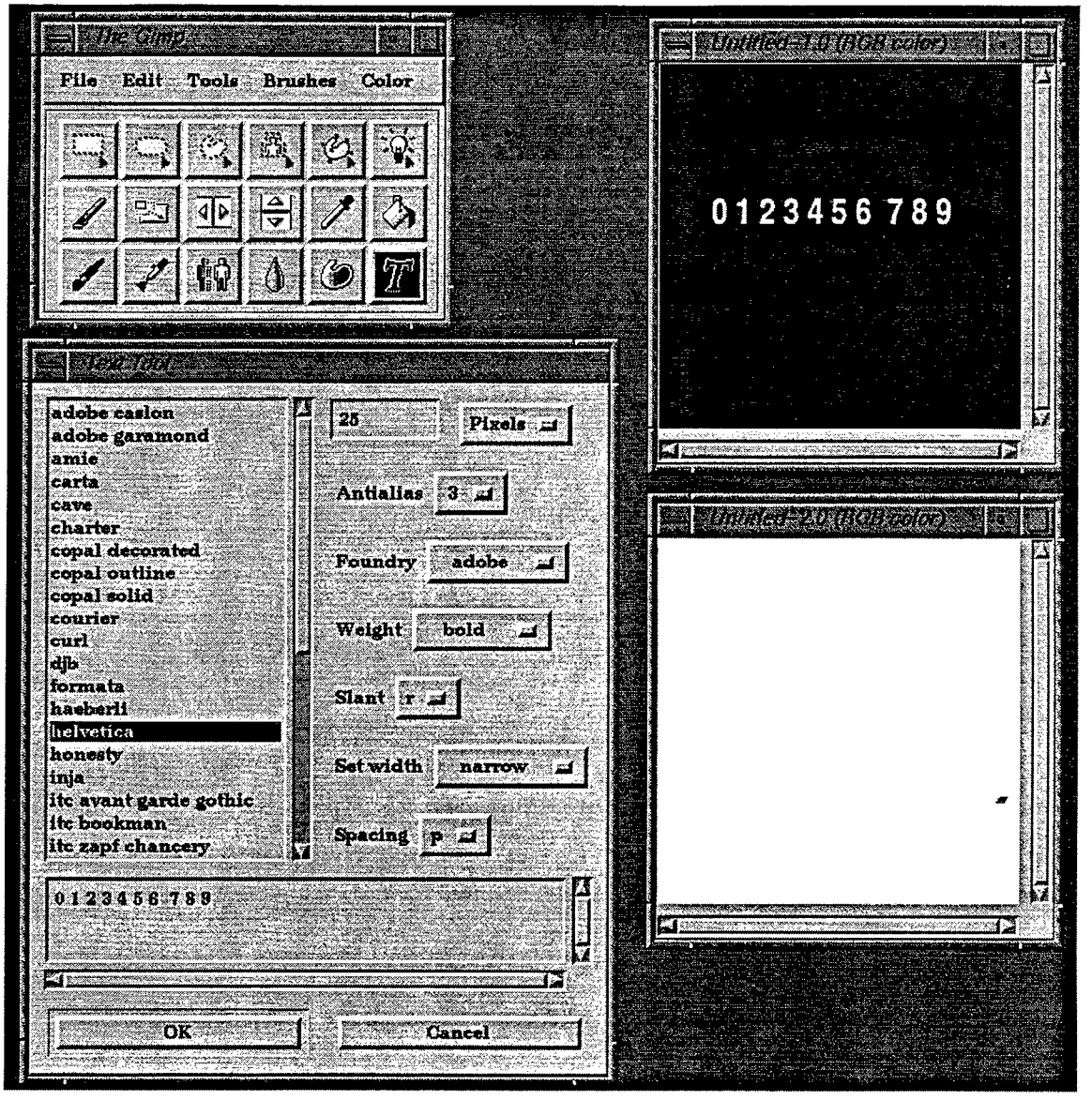

Figure 4-4. Creating A Second Image That Is Entirely White (0).

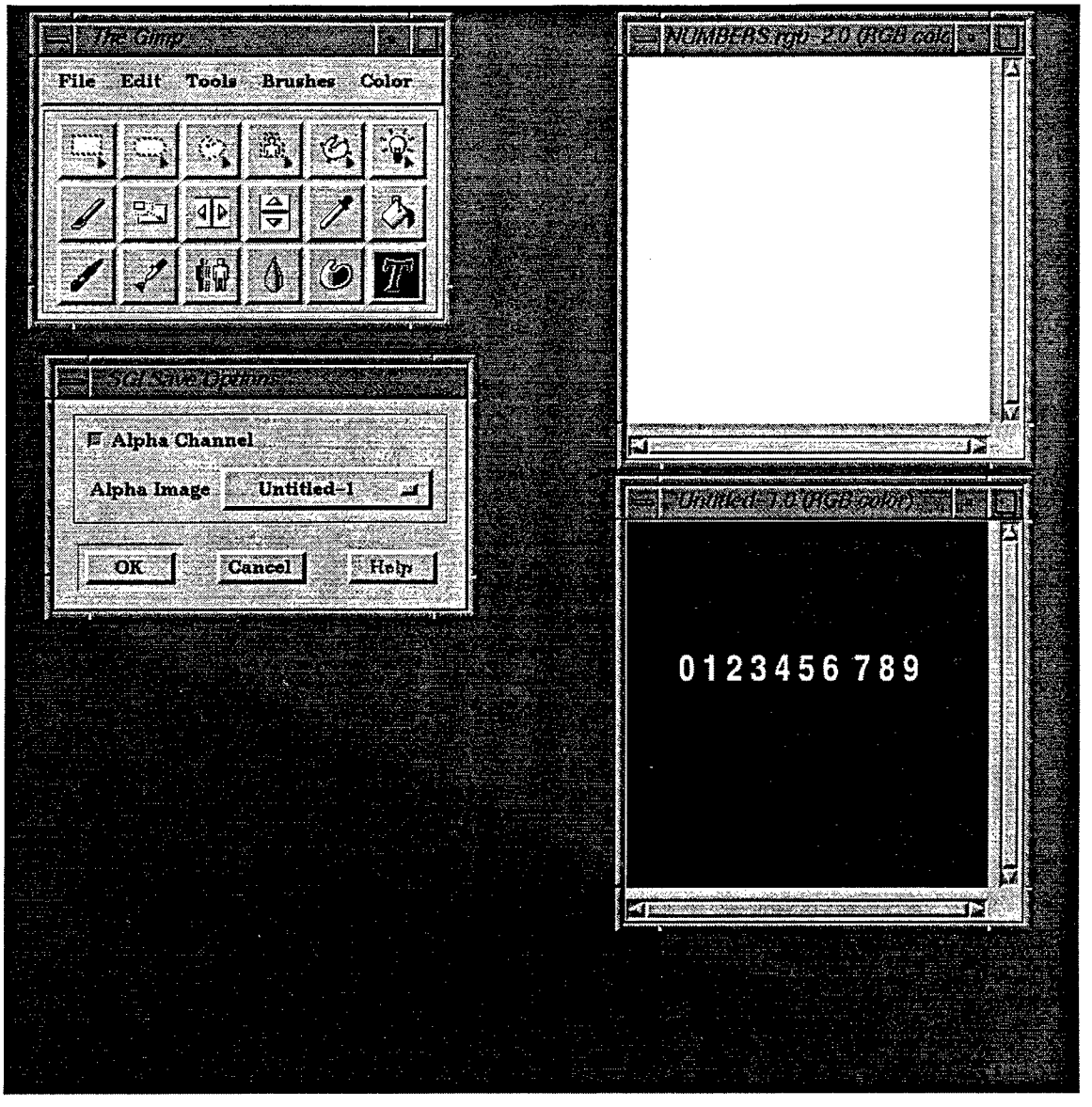

Figure 4-5. Saving Second Image As An RGBA File Using The First Image As The Alpha Channel.

Note: The reason all the number textures were created in blocks is because Performer needs a texture's dimensions to be a power of two. Creating several lines of text allowed more text to be placed in a power-of-two texture. The textures could then be placed on an identically proportioned polygon in DWB and individual words / phrases cut out using the polygon cutting tool.

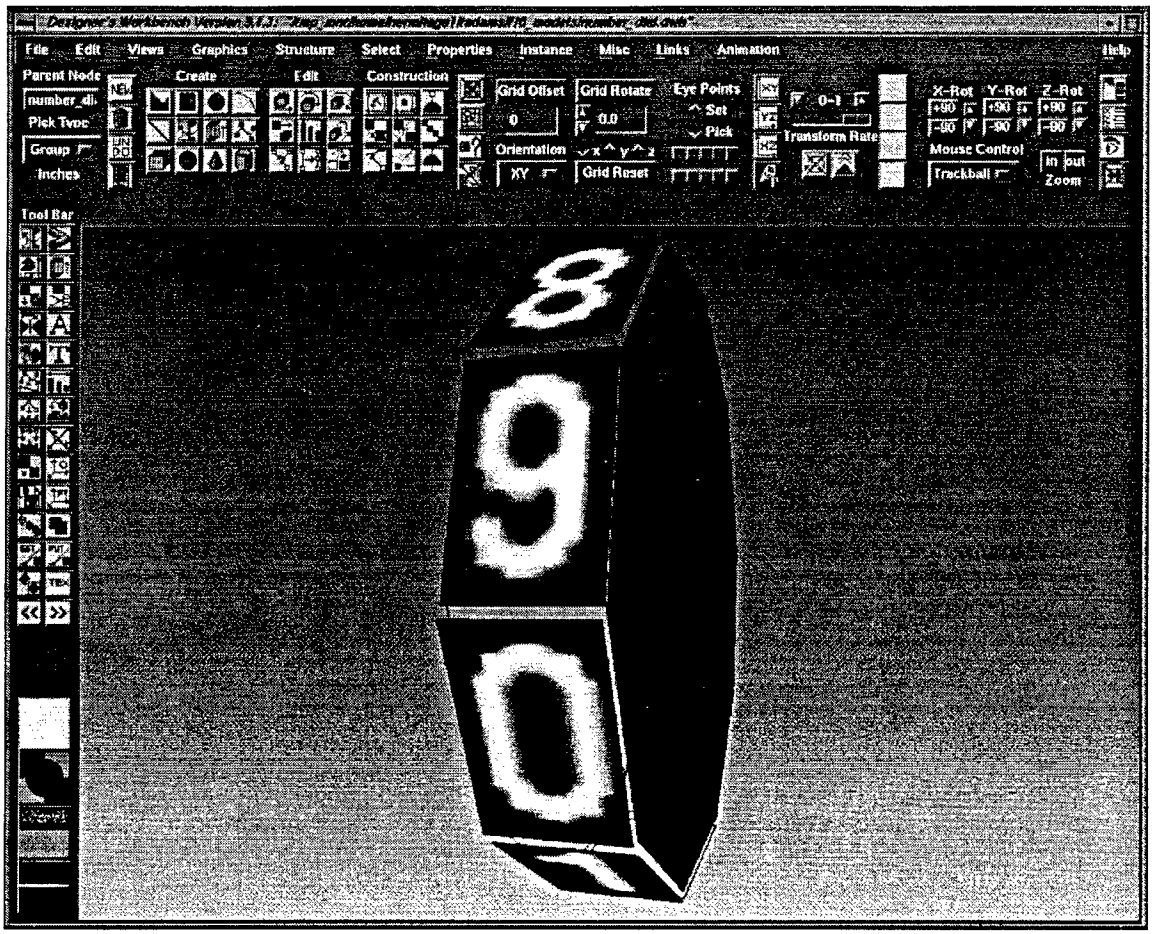

Figure 4-6. Single Altimeter Dial in DWB.

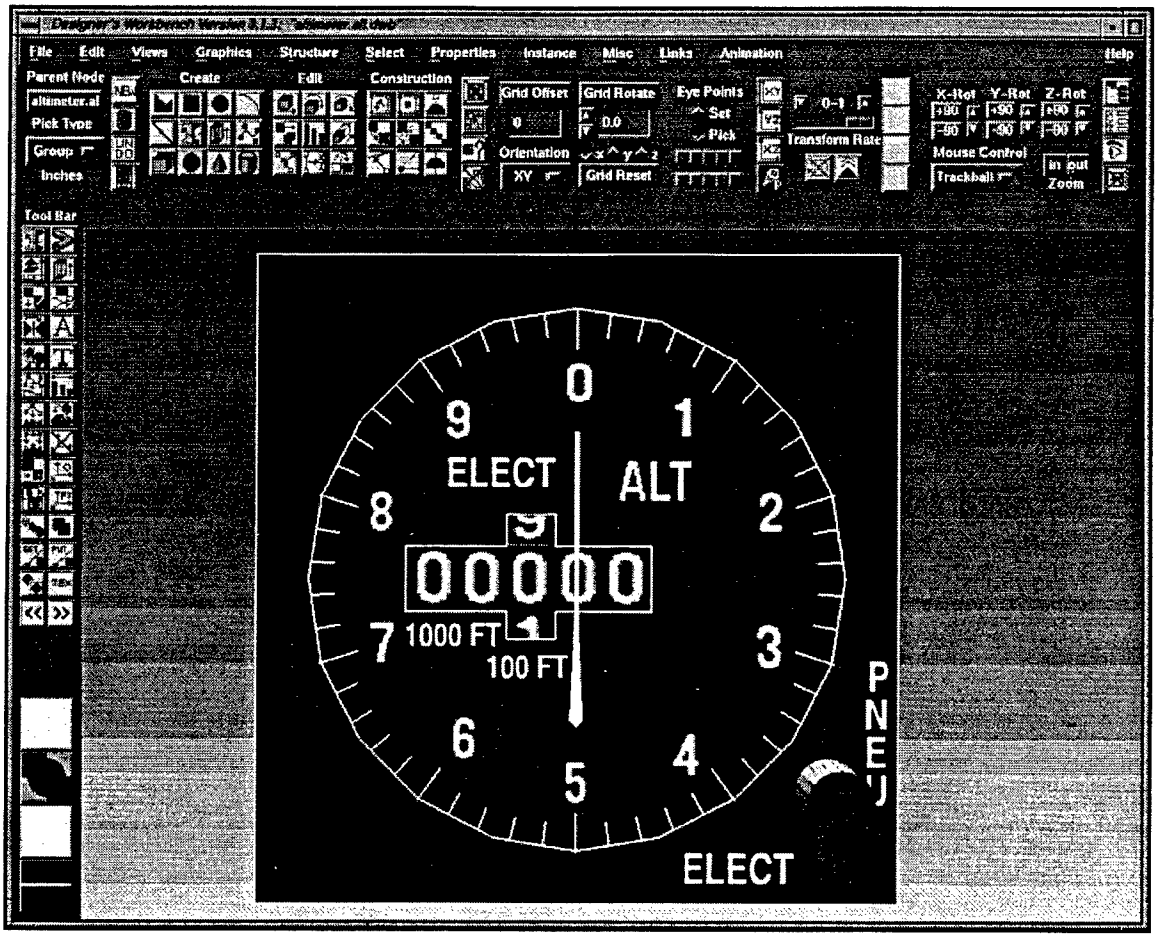

Figure 4-7. Entire F-16 Altimeter in DWB.

Once the texture is completed, the rest of the altimeter's geometry can be created. Number dials are created using the number texture (see Figure 4-6). Creation of individual number dials allow simulation of individual spinning dials and gives an impression of how fast the aircraft is changing altitude. Spinning dials could be simulated using GL calls, but would have used a vector font for the dials that is not as realistic as the texture-mapped dials. To model the performance of the actual altimeter in Performer, each component that moves independently must be stored in an individual file. The altimeter includes several independent components: the needle, a number dial (that is duplicated six times), a switch, a

PNEU/ELECT label, and the background (that does not move), see Figure 4-7. These individual files were each developed by creating an entire altimeter file and then saving each of the above components in a file by itself. The background model is then added at the desired location in the Performer simulation with all of the other components in the same Performer tree, Figure 4-8. Movement of the dials and needle is accomplished by rotations based on the current speed of the aircraft. Movement of the PNEU/ELECT label is in response to a user moving the switch (switch movement is covered in a later section of this chapter) and also involves simple rotations based on switch position. All updates of dials and switches are accomplished in the Update method of the f16_instruments class that also contains the code necessary to load all of the cockpit geometry.

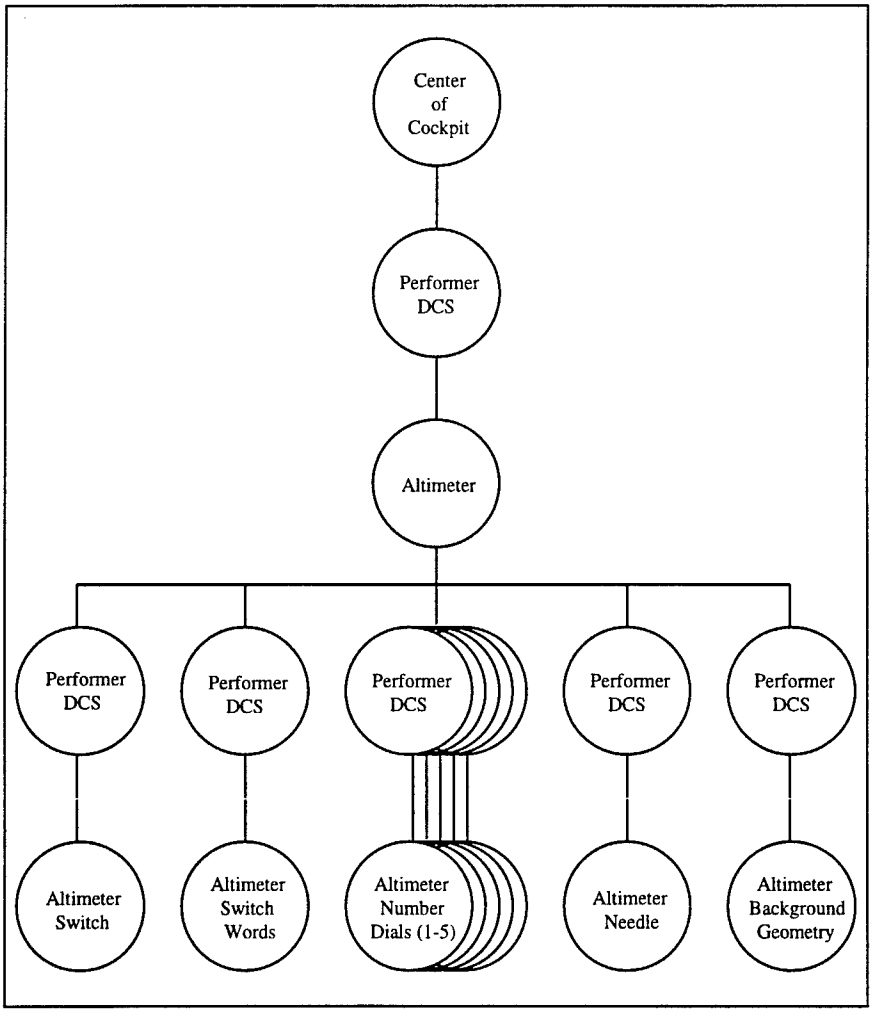

Figure 4-8. Altimeter Performer Tree

Each of the instruments in the F-16 cockpit is created in the same manner as above. The aircraft model is used to drive all of the displays. However, some displays, such as oil pressure and exhaust temperature are not a direct output of the aircraft model and had to be derived from throttle position [ULST96a]. Pictures of all the F-16 VC instruments are provided in Chapter 5.

4.3. Reconfigurable Simulation Components

Reconfigurable software models provide a single point solution for multiple modeling situations. The primary reconfigurable model utilized here is a reconfigurable aircraft model that can represent any modern aircraft based on a series of input files. Secondary reconfigurable models included a simple radar model and the modification of the weapons controller to allow multiple aircraft to use it independently.

The aircraft model that was previously used in the VC did not provide enough fidelity for Virtual Reality simulation, often visibly shaking or jittering during flight. A reconfigurable aircraft model was obtained from Wright Laboratory's Flight Simulation Branch (WL/FIGD). WL/FIGD obtained the model under a contracted simulation effort with Eidetic. The aircraft model is protected from public release; but, is releasable to any government entities who wishes to use it. The model was made up of several C language files and allowed the creation of one aircraft that would take on the characteristics of the aircraft specified in the input files. The model provides no support for Performer, CODB, changing aircraft, or changing aircraft position. Separate implementation phases were used to change the C model into one that the RRVC could use for in-flight reconfiguration and are as follows:

- change C language model into C++ to allow multiple aircraft to be created,
- use CODB for input and output and provide Performer support, and
- allow in-flight reconfiguration of aircraft model.

The phases were accomplished in the order above and are discussed in detail in the following paragraphs.

The change from C to C++ was primarily done to allow multiple aircraft models to be created and not to restructure or redesign the model itself. Therefore the model was simply encapsulated in a C++ wrapper. All C variables were made private instance variables of the class. All C routines were made

private methods in the AeroModel class. A set of public instance variables and methods were created to interface with the private low-level aircraft model software. These variables and methods insulate the developer from the low-level aircraft model and provide an easy to understand interface, Figure 4-9. Reset and Fly methods are consistent with the underlying C model. Additional methods, developed for C++ included a Constructor, Get and Set routines for all inputs to the aircraft model, and only Get routines for all outputs of the model. A problem uncovered during use of the new model for multiple aircraft was that identical throttle values were being used for both instantiations of the aircraft regardless of how they were set in the methods. This problem points out a critical difference between C and C++, how they treat static variables. A static variable in a C function is created and initialized once regardless of how many times the procedure is called. In a C++ class, only one copy of the variable is created for all instantiations of the class. The CODBAeroModel's error involved the use of a static variable to filter past throttle values and was being shared among all instantiations of the class. This problem was the result of an oversight during the conversion process and was quickly corrected.

Making the AeroModel into the CODBAeroModel class required adding CODB accesses and providing Performer support. Instead of overloading AeroModel methods, a new class was created that inherited behavior from the AeroModel Class. This allows future developers to choose between a CODB aircraft model and a non-CODB aircraft model. The new CODBAeroModel interface is shown in Figure 4-10 and provides additional methods beyond the basic AeroModel Class. A link to the CODB was added to the constructor of the class to provide CODB access. The constructor was also overloaded to allow the developer to choose the aircraft the model would emulate. In addition an AircraftStruct was added to the CODB control structure to store the aircraft model's output. The decision was made to only place in the container information that is required by other simulation entities to limit container size and reduce complexity. The container entries can be seen in Figure 4-11. Input to the model is provided by the IO_Hotas class via the HotasStruct CODB structure. The Fly method is responsible for all input and output processing. Input is handled at the entry to method with the AircraftStruct CODB container updated

```
//Constructor
AeroModel(); //defaults to trimmed and F16
AeroModel(AircraftType ac_type, int trim_aircraft);

//Methods to get input parameters
float GetStickLateral();
float GetStickLongitudinal();
float GetThrottle();
float GetRudderPedal();
int   GetSpeedBrakeCommand();
float GetThrustReverser();
int   GetPiloted();
float GetPitchTrimCommand();
float GetRollTrimCommand();
float GetYawTrimCommand();

//Methods to set input parameters
void SetStickLateral(float lateral);
void SetStickLongitudinal(float longitud);
void SetThrottle(float throt);
void SetRudderPedal(float rp);
void SetSpeedBrake(int asb);
void SetThrustReverser(float tr);
void SetPiloted(int p);
void SetPitchTrimCommand(int trim);
void SetRollTrimCommand(int trim);
void SetYawTrimCommand(int trim);

//Reset the aircraft to position specifiec
//Retained firstfunction for compatability can be
replaced by next function

void ResetPosition(float acft_x, float acft_y, float acft_z,
                   float acft_mach, float acft_psi);
void ResetPosition(float acft_x, float acft_y, float acft_z,
                   float acft_psi, float acft_theta,
                   float acft_phi, float acft_mach);

//Fly aircraft one time step
void Fly();
```

```
//Methods to set input parameters
float GetNorthPos();
float GetEastPos();
float GetDownPos();
float GetTheta();
float GetPhi();
float GetPsi();
float GetAlpha();
float GetBeta();
float GetMach();
float GetGLoad();
float GetAirspeed();
float GetGamma();
float GetSigma();
float GetMu();
float GetSpeedBrakeAngle();
float GetVelocity();
float GetRollRate();
float GetPitchRate();
float GetYawRate();
float GetXLoad();
float GetYLoad();
float GetZLoad();
float GetRho();
float GetFuelFlow();
float GetFuelAmount;
float GetAircraftWeight();
float GetSustainedLoadFactor();
float GetMaxInstantLoadFactor();
float GetThrust();

float GetDrag();
float GetAircraftStructuralLimit();
float GetMaxAOA();
float GetMinAOA();
float GetXVelocity();
float GetYVelocity();
float GetZVelocity();
float GetXAcceleration();
float GetYAcceleration();
float GetZAcceleration();
float GetRollAcceleration();
float GetPitchAcceleration();
float GetYawAcceleration();
```

Figure 4-9. C++ AeroModel Interface

```
CODBAeroModel(); //defaults to F-16 and trimmed
CODBAeroModel(AircraftType ac_type, int trim_aircraft);
void ResetPosition(float acft_x, float acft_y , float acft_z,
                   float acft_mach, float acft_heading);
void ResetPosition(float acft_x, float acft_y, float acft_z,
                   float acft_heading, float acft_pitch, float acft_roll,
                   float acft_mach);
void Fly();
void GetPerformerPos(float& x, float& y, float& z);
void GetPerformerEuler(float& h, float& p, float& r);
void GetPerformerLinearVel(float& x, float& y, float& z);
void GetPerformerLinearAcc(float& x, float& y, float& z);
void GetPerformerAngularVel(float& h, float& p, float& r);
void GetPerformerAngularAcc(float& h, float& p, float& r);
```

Figure 4-10. C++ CODBAeromodel Interface

after the aircraft has flown out a single time step. Performer support was provided by using Performer coordinates in the AircraftStruct and creation of methods to provide Performer coordinates, orientations, velocities, and accelerations. The changes required between the Aircraft Model's flat earth coordinates and Performer's flat earth coordinates are discussed in the Zurita reference [ZURI96].

```
float x, y, z;                  //degrees
float h, p, r;                  //degrees
float vel_x, vel_y, vel_z,      //meters/sec
float vel_h, vel_p, vel_r;      //degrees/sec
float mach_speed;               // mach
float thrust;
float fuel;                     //lbs
float speed;                    //in knots
float rpm;
float left_rpm;
float right_rpm;
float g_load;
float altitude;                 //meters
float v_velocity;  //meters/sec
float angle_of_attack;          //in degrees
float beta;                     //in degrees
float speed_brake,              //in degrees
int afterburner                 //0 = off, 1 = on
```

Figure 4-11. CODB AircraftStruct Container

The final step in making the aircraft model reconfigurable was changing the ResetPosition method of the CODBAeroModel class to allow any orientation and speed. When a user changes the aircraft type by switching CODBAeroModels in the Airplane class they need to begin flying the new model in the same orientation as they were last oriented. The C aircraft model only supported resetting the x, y, and z position along with the aircraft's heading and mach number. A mach number that was insufficient to fly straight and level would cause the software to core dump and quit executing. The constructor of the aircraft model was changed to allow the user to specify whether straight and level flight was a prerequisite for the model's initialization. The code that enforced this prerequisite was then enclosed with an if-then statement based on the value passed into the constructor. In addition, modifications were made to the original C ResetPosition function to allow the developer to set the entire orientation of the aircraft, heading, pitch, and roll. When the developer calls Reset_Aircraft_Type to F-16 in the Airplane class, the model is switched from a F-15 CODBAeroModel to a F-16 CODBAeroModel. This is followed by a call to CODBAeromodel:: ResetPosition with the last position and orientation of the F-15 CODBAeroModel as the parameters.

```
simple_radar();
~simple_radar();
simple_radar(float min_relative_azimuth,
             float max_relative_azimuth,
             float min_relative_elevation,
             float max_relative_elevation,
             float max_range_miles);
void get_azimuth(float& min_azimuth, float& max_azimuth);
void set_azimuth(float min_azimuth, float max_azimuth);
void get_elevation(float& min_elevation, float& max_elevation);
void set_elevation(float min_elevation, float max_elevation);
void get_range(float& range);
void set_range(float range);
//Check all entities in LocalCoordStruct for Radar visiblility
//update RadarStruct appropriately based on radar field of view.
void update();
```

Figure 4-12. Simple Radar Model Interface.

A second reconfigurable model was created to simplify the current radar model and allow customization of the radar for several different aircraft. The 1995 VC's radar model was hidden inside the INS class and provided no methods to set radar parameters. Additionally, the model only provided a visible

/ not visible designation and no other radar parameters, such as azimuth, elevation, and range. A diagram of the new reconfigurable model with its interface is shown in Figure 4-12. The new radar class uses the LocalCoordStruct container (contains all DIS entities, in flat-earth Performer coordinates) from the CODB as input and places its output in the RadarStruct container. Both the LocalCoordStruct and RadarStruct containers are shown in Figure 4-13. All active tracks in the LocalCoordStruct are evaluated for their relative azimuth, elevation, and range to the RRVC. If the tracks are within the radar's field of view then track_state is set to Active for that track in the LocalCoordStruct, if not the track_state is set to Inactive. Figure 4-13 shows the structure of the input and output containers which allow correlation between them. The radar track is in the same position in the RadarStruct as it is in the LocalCoordStruct to allow correlation by other RRVC components if needed. Active track's position information is also inserted into the container. The model is easily reconfigurable by using the Set methods to fit the specific aircraft's field of view. and will provide a baseline for more complicated models.

Implementation of a reconfigurable weapons controller was much simpler than the aircraft or the radar models. The weapons controller class provides an interface to cannon, bomb, and missile classes for the simulation. The weapons class uses a file to initialize the number of weapons and their position. Unfortunately, this only allowed one weapons controller because the filename was hard-coded into the class. The init method of the weapons controller class was changed to include a character string that indicates the file to be used for weapons initialization. In addition, the maximum number of each type of weapon allowed by the class was increased to allow weapon load outs associated with multiple aircraft. The changes were simple, but needed to allow different aircraft to use the same weapons controller class to control their weapons.

Background Structures

```
struct entity_appearance_record {
  // Entity Location in DIS coordinates
  double        x;          // meters
  double        y;          // meters
  double        z;          // meters

  // Entity Orientation in DIS coordinate system
  float         psi;        // radians
  float         phi;        // radians
  float         theta;      // radians

  // Entity Linear Velocity vector in DIS coordinates
  float_vector linear_velocity;   // m/sec

  // DIS ID
  unsigned int      site_id;
  unsigned int      application_id;
  unsigned int      entity_id;

  // Local Use Only
  unsigned short    model1;
  unsigned short    model2;

  // Entity Description
  char          description[40];   // ASCII characters

  // Force Identifier
  entity_alliance   team;

  // Enumerated type to determine type of entity
  container_state   entity_state;

} entity_appearance_record;
```

LocalCoordStruct

```
typedef struct entity_appearance_container {
entity_appearance_record
DIS_entity[MAX_NUMBER_OF_ENTITIES];
  unsigned int num_of_active_entities;
} entity_appearance_container;
```

Background Structures

```
struct radar_entity_struct {
//Entity's Radar Orienation and Position
float x,
float y,
float z,
float h,
float p,
float r,
float az,
float el,
float range;
container_state track_state;
};
```

RadarStruct

```
struct radar_container {
  radar_entity_struc
    radar_entity[MAX_NUMBER_
                 OF_ENTITIES];
  int number_active_tracks;
  float min_relative_azimuth, //degrees
  float max_relative_azimuth; //degrees
  float min_relative_elevation, //degrees
  float max_relative_elevation; //degrees
};
```

Figure 4-13. Comparison of LocalCoordStruct and RadarStruct

4.4. Replacing ObjectSim Functionality

Getting rid of ObjectSim in the VC impacted three areas of the implementation: changing the viewpoint in the simulation, drawing entities, and interacting with the DIS components. Each of these areas of functionality needed to be duplicated in the VC without using the ObjectSim framework. The viewpoint in the VC is aircraft centered at the origin, due to Performer limitations, and was managed by an ObjectSim class. The creation of a structured Performer entity tree is needed that will perform the entity management

task. Finally, interaction with DIS was encapsulated within ObjectSim and must be moved to the CODB architecture (the DIS interface will be discussed in a following section of this chapter). These areas of functionality had to be implemented to maintain the functionality and performance associated with the 1995 VC.

The viewpoint must be centered at the origin because of Performer limitations. So not only must it be aircraft centered, this is expected for an aircraft simulation, but, the aircraft must be also centered at the origin. This is because Performer has limited resolution for viewpoint position and as the viewpoint begins to get further away from the origin it begins to shake or wobble as it loses the resolution necessary to maintain a steady viewpoint. This behavior is apparent in any large virtual Performer environment such as the VC where viewpoint resolution is important. The problem was not apparent in the baseline CODB application that moved the viewpoint around the world with the aircraft because the viewpoint was above and behind the aircraft. When the viewpoint was moved inside the cockpit, where small movements in viewpoint make all the geometry shake around, the problem was readily apparent. The problem was fixed using an algorithm from the ObjectSim View class (algorithm provided in Chapter 3 [SNYD93]. The algorithm moves the geometry of the aircraft to the origin and then moves all the rest of the world (DIS entities and terrain) relative to the aircraft based on its position and orientation. Use of the algorithm requires the Performer tree to be structured in such a way that all geometry can all be moved relative to the aircraft. The algorithm is now in the VC_Renderer class in the Make_Final_View method.

The Performer tree implemented in the VC_Renderer has the same structure as the ObjectSim tree structure. Figure 4-14 shows the top-level Performer tree's structure for RRVC. The tree has three subtrees: DIS entities, RRVC aircraft, and Terrain. The pfDCS nodes above each of the subtrees allow the Terrain and DIS entities to be manipulated by the Make_Final_View method, while the pfDCSs in the Players subtree allow manipulation of the individual players. The pfSwitch node was added in the VC subtree to allow switching between different aircraft geometries. Maintaining the current structure also eased the integration of the weapons classes, that move weapon models from the RRVC subtree when they are attached to the DIS entities subtree once they have been fired.

4.5. Replacing AFIT Pod Interface

Replacement of the AFIT Pod Interface focused on reducing the complicated nature of the AFIT Pod and on allowing natural interaction with the RRVC virtual environment. The AFIT Pod consists of several classes that must be inherited from and components of that must be over-ridden resulting in a complicated class that is difficult to use. In addition, the Pod only allows interaction with buttons set on a flat rectangular surface. To expand the interaction capability a class that encapsulates Performers built in geometry picking functions was developed by Captain Brian Garcia. The class provides a way of determining if a simulation object was selected by the user or if a portion of a larger object was selected. By using geometry, the class is able to accurately pick irregularly shaped objects instead of always picking against a square area as in the AFIT Pod. The RRVC research project extended and modified the basic Selection Manager class to maintain a consistent visual interface with the 1995 VC and to allow a developer to customize their interaction techniques.

The basic class is simple to use and requires minimum developer interaction. The class requires the user to name all of the selectable Performer nodes. The user calls the Selection Manager class constructor, passing it the top of the Performer tree to be selected and the Performer channel where the tree will be displayed. These actions set up the Performer tree for selection. The class works by getting the path through the tree of the node selected and the poll method returns an integer that indicates the selected node. For instance, by naming the altimeter node NODE_0001 the poll method will return a one (1) to the calling routine when that node is selected. The class was extended to allow multiple level of nodes to be selected. A user can use the SetSelectionLevel method to determine which level interests them (i.e., the instrument panel (level 0), the altimeter on the instrument panel (level 1), or the switch on the altimeter (level 2)). The model was also expanded to allow the user to determine what was selected at any level of the Performer tree and to allow node selection with any mouse button. A final addition to the class allows the user to pass a user-defined segment into the class for picking purposes in the poll method. The process may seem complicated, but it is greatly simplified from the AFIT pod protocol (see Figure 4-15 for a comparison of what is needed to update a button in each protocol). The figure is needed for comparison purposes and

illustrates a basic button interaction. The actual instrument panel buttons are more complicated because they use Performer geometry for the buttons instead of the default GL buttons provided by the button_type class. The F-15 Pod interface has an additional class that is responsible for maintaining the state of this geometry based on the GL buttons. The new Selection Manager does not have any default buttons and uses only user-defined geometry for buttons, which is the more complicated case.

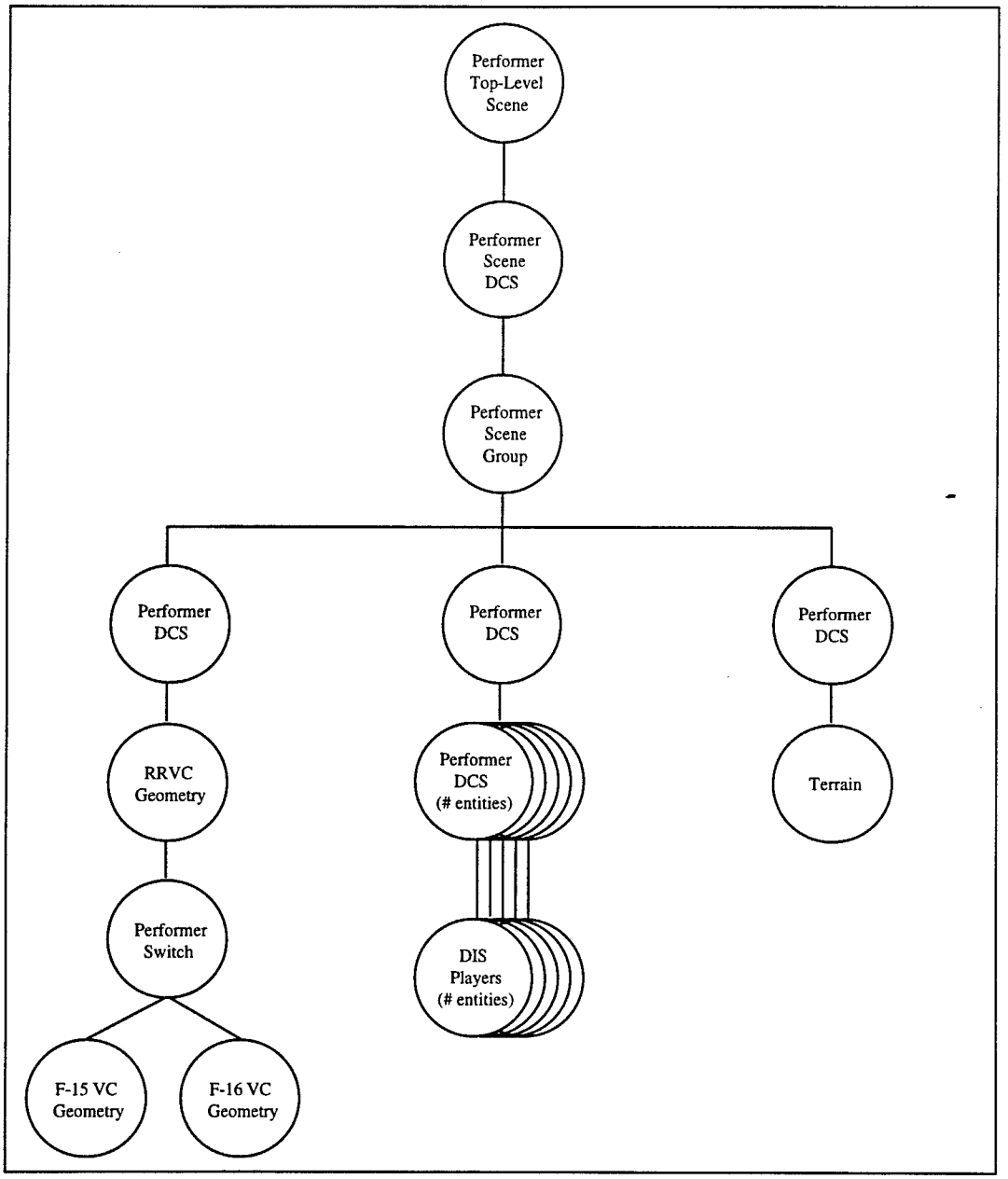

Figure 4-14. RRVC's Top-level Performer Tree.

AFIT Pod	New Selection Manager

Create class inheriting from Panel_Type including the following virtual functions:
-- Sets state information for all sub-panels on panel
void Update_Children (pfSeg* Segment, int MouseButtonStatus, pfVec3 X_Finger_Marker)
-- Returns "hard-coded" vertex information for the button
void Get_Point_1 (pfVec3& Point)
void Get_Point_2 (pfVec3& Point)
void Get_Point_3 (pfVec3& Point)
void Get_Point_4 (pfVec3& Point)
--Sets up the pre- and post- callbacks for the panel
void Register_Callbacks ()
--Initializing of any buttons, etc. that belong to this panel
void Initialize_Children ()
The panel class should have as an instance variable a sub-panel that must be of a class inheriting from Sub_Panel_Type including the following virtual functions:
--Initializing of any buttons, etc. that belong to this panel
void Initialize_Children ()
-- Sets state information for all buttons on sub-panels
void Update_Children (pfSeg* Pointer_Finger, int MouseButtonStatus, pfVec3 X_Finger_Marker)
-- Draws the GL buttons if required
void Draw_Children()
The newly derived sub-panel class has an instance variable of Button_Type that defines button size, orientation, color on/off, text, and arrow orientation (if desired).
 	
The variable of newly derived panel class is created, that also creates a sub-panel and a button.
 	
A variable of type mouse is created.
 	
The panel is then added to the mouse with the method Add_Panel that describes the panel's position and orientation.
 	
The panel is made active with the method panel.Set_State. If more than one panel is created the user must use the right mouse to switch between panels.
 	
During each time step of the application the following calls must be made:
The method mouse.propagate is called during each cycle of the application to check if a the mouse has moved and if a button was pressed (to change panel of interest)
--Returns starting and ending coordinates for a segment beginning at cursor and going through the panel of interest.
mouse.Get_Pointer_Info (Start_X, Start_Y, Start_Z, End_X, End_Y, End_Z);
--Convert pointer xyz coordinates into pfseg
PFSET_VEC3 (Start_Point, Start_X, Start_Y, Start_Z);
PFSET_VEC3 (End_Point, End_X, End_Y, End_Z);
pfMakePtsSeg (Segment, Start_Point, End_Point);
--Check to see if the user is depressing the left mouse button.
Mouse_Left_Button = Mouse.Get_Mouse_Left_Button_Status();
--Check mouse.Current_Panel to see that panel is active (in this case there is only one panel) and call the update for that panel. This call will also update the state of any buttons that were pressed.
panel.Update (Segment, Mouse_Left_Button);
	Create and Initialize the Selection Manager.
	
	--Add the button geometry to the Performer tree and name the nodes to be selected.
	pfNodeName(Altimeter_Switch, "NODE_0001");
	
	--Create a Selection_Manager passing the Performer subtree that can be selected, the channel the tree will be displayed on, and the key string to be searched for (in this case "NODE_)
	Selection_Manager(pickable_tree, channel, "NODE_");
	
	During each time step of the application the following calls must be made:
	
	--Poll the Selection_Manager
	button_pressed = Selection_Manager.poll;
	
	-- If button pressed = 1 then the Altimeter_Switch has been pressed and the state of the button can be updated by translating or rotating the geometry as desired.

Figure 4-15. Comparison of AFIT Pod Interface and Selection Manager Interface.

The additional capabilities provided by the Selection Manager for the F-16 cockpit provide interaction techniques that are more dynamic then those in the F-15 cockpit. The Selection Manager was not integrated into the F-15 cockpit because of the time associated with the change and it provides a good contrast between the two selection methods. To move a switch that has a rotation range of +/- 90 degrees, the F-15 cockpit required right mouse clicking on the button to move it clockwise one setting. The user could not move the button counter-clockwise until the button had reached its maximum clockwise rotation. If a button has six settings and was on the second clockwise setting the user had to press the button four more times to cause the switch to reset at its minimum clockwise rotation. The change from moving one setting at a time in the clockwise direction to suddenly jumping from setting six (maximum clockwise rotation or 90 degrees rotation) to setting one (minimum clockwise rotation or -90 degrees rotation) is unnatural to the user. This is unnatural because it is not how the switch actually reacts in the airplane. The new selection manager allows a user to select cockpit switches or dials with multiple mouse buttons. This capability provides the developer with the ability to move switches counter-clockwise if the left mouse button is pressed and clockwise if the right mouse button is pressed. The switch stops at both its minimum and maximum settings with no unnatural jumping. This ability also allows the developer to create dials that move left or right based on mouse input and the F-16 VC uses the middle mouse button to rotate the dial faster in either the left or right direction. The Selection Manager is easily customizable and allows developers greater flexibility in developing geometry selection programs. All new development in the AFIT VC should make full use of this class to ease development and to examine more natural cockpit interaction techniques.

4.6. Distributed Simulation Interface

The RRVC DIS simulation interface is through the World State Manager 3.0 (WSM). WSM provides both PDU send and receive capability using the CODB architecture. The World State Manager provides containers for broadcasting aircraft and weapons entity state PDUs and for receiving entity state PDUs. The structures contain similar data but have a different structure as can be seen in Figure 4-16.

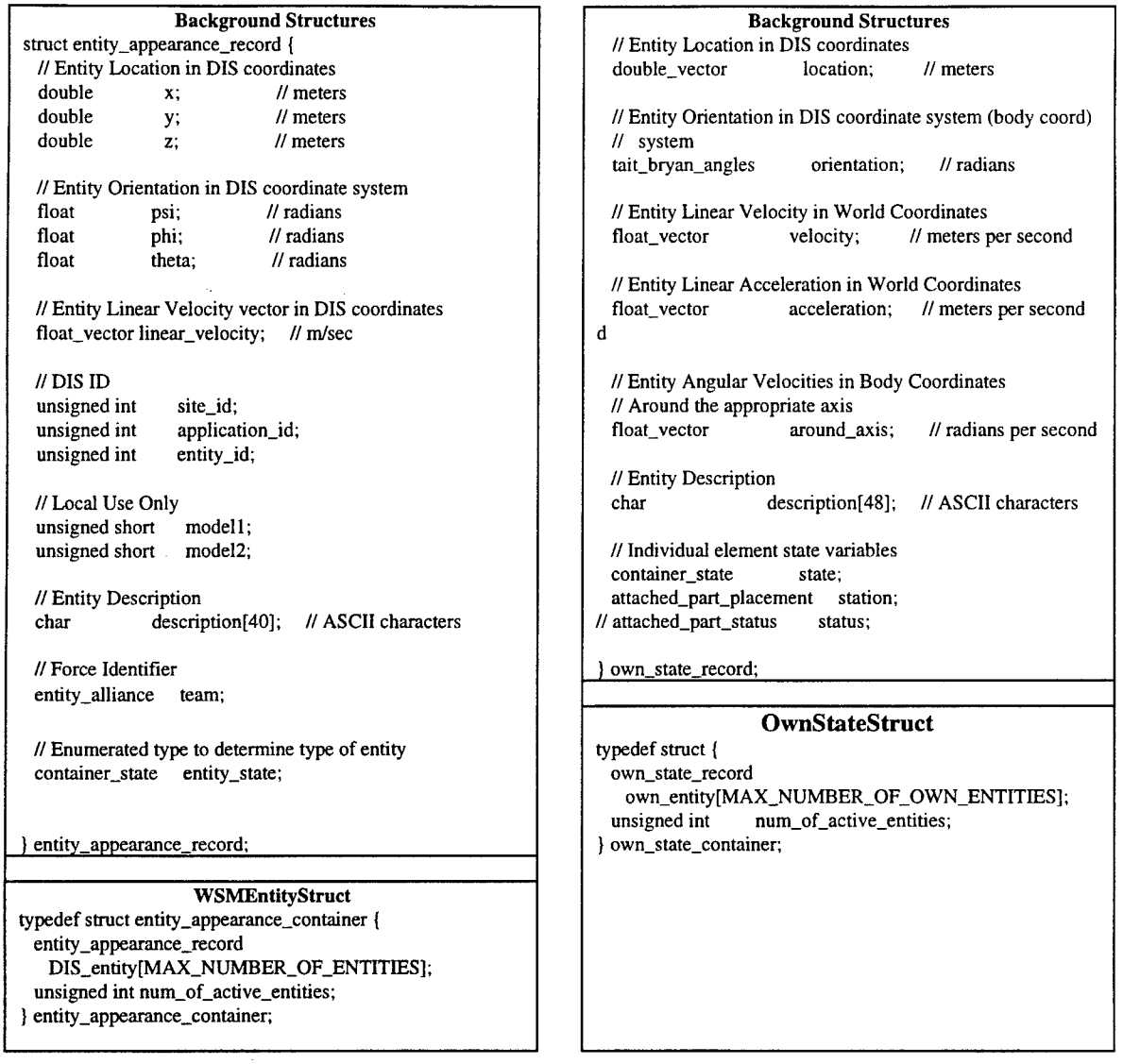

Figure 4-16. Send and Receive DIS CODB Containers.

The WSM uses these structures to provide a DIS capability to the RRVC, allowing it to take part in distributed training exercises. The RRVC uses the entity information provided by the WSM to update its own representation of the virtual environment. The RRVC sends out entity state information to other entities on the network indicating its state information, including the type of aircraft it represents and its position and orientation. Implementation of the RRVC's DIS interface was to be broken down into two distinct steps: sending out entity state PDUs and receiving entity state PDUs. However, due to limited

functionality of the initial World State Manager, the sending out of weapons entity state PDUs was delayed and became a third step to the DIS integration process.

Receiving entity state information is the most complex portion of the DIS interface because the entities must be converted into the local simulation's world coordinate system and then displayed accurately to the pilot. This follows a four step process:

1. Read WSMEntityStruct container from CODB,
2. Convert DIS coordinates into Performer flat-earth coordinates,
3. Store coordinates in LocalCoordStruct (container identical to WSMEntityStruct) container in CODB, and
4. Update position and orientation of aircraft in Performer geometry.

A library function, Convert_To_Local, was created to take in a pointer to a LocalCoordStruct, WSMEntityStruct, and to a RoundEarthUtils class. The RoundEarthUtils class is a class used to convert between DIS coordinates and Performer flat-earth coordinates. The Convert_To_Local function reads the data from the WSMEntityStruct container, converts it using the RoundEarthUtils class, and the stores it in the LocalCoordStruct container. The function insures that all data from the DIS container is placed in the local container. If any items are added or removed from the WSMEntityStruct container this function must also be updated to reflect those changes. The VC_Renderer class method Update_Players uses the LocalCoordStruct container to update the position and orientation of the Performer geometry that represents the aircraft.

Sending entity state information is easy with the WSM. The Airplane class broadcasts its positioning by placing its entity state information into the own_state_record in position zero of the own_entity array in the OwnStateStruct. The coordinates must be converted from Performer flat-earth to DIS coordinates for transmission. RoundEarthUtils performs this task as it did in the Receive situation. The description field of the container is updated to reflect the current aircraft being modeled by the RRVC. Finally, a call is made to the WSM indicating position zero holds information of interest. The structure is

updated every frame of the simulation. The WSM uses dead-reckoning to determine when to send out a packet on the network, without any application interaction.

The final step of DIS integration was sending out entity state information for RRVC weapons. The same process is followed by the weapons for sending out entity state information; however, it only happens after the weapon has been dropped or fired. To indicate a weapon has been fired a Fire PDU must be sent out on the net. The WSM sends out a Fire PDU after being notified by the function broadcast_weapons_fire. The WSM uses the index passed to broadcast_weapons_fire to determine the position in the OwnStateStruct container where the weapon's information is stored. A Detonation PDU indicates the explosion of a weapon and is handled in the same manner, with a call to broadcast_weapons_detonation. The index used to determine the position of the weapon in the array is determined at initialization time, depending on the number of available weapons. The RRVC uses indexes from ten to the maximum index minus one to store weapon state information. The maximum index position is saved for bullet information that is communicated only with Fire and Detonation PDU (no entity state PDUs in between). The weapon models had already been interfaced to the old WSM, so the determination of when certain PDUs should be sent was already available. However, instead of having each weapon interfacing directly to WSM (as it was in the 1993 VC when weapons were last available), the DIS interface is primarily encapsulated in the WEAPONS_CONTROLLER class.

4.7. Conclusion

The development of the RRVC has shown that a cockpit can be rapidly reconfigured between two separate aircraft, a F-15 and a F-16. Each stage of the implementation provided a feature to the VC that had not existed before. Implementation of the CODB architecture allows new multiprocessor simulation components to be created without worrying about how memory access takes place. The entire F-15 VC was reimplemented using the new architecture and all ObjectSim code was removed. A new F-16 VC was developed and integrated into the new architecture providing a reconfigurability testbed. Several new components were developed to provide a sample simulation interface, including a simulation rendering component, aircraft model, radar model, cockpit model, and new geometry selection manager. The F-16

was built upon these components that provide both a sample and basis for future expansion of the RRVC. A new World State Manager based on the CODB was integrated into the simulation to provide an interface to other DIS environments. The final result is a Rapidly Reconfigurable Virtual Cockpit that can take part in DIS exercises as either a F-15 or a F-16 or both.

5. Result

This chapter discusses the results associated with each area of the RRVC's design. This section refers back to the requirements and goals discussed in Chapter Three, primarily Table 3-1. For convenience, Table 3-1 is broken into sections and provided in Tables 5-1 to 5-6. The results in all areas were very favorable. The RRVC was able to successfully simulate both an F-16 and F-15 using the CODB architecture. The RRVC was able to switch between aircraft models in a single frame. With a frame rate between 12 frames to 15 frames per second, the time required to switch cockpits was less than 0.1 seconds. DIS support was provided by the CODB-based World State Manager and allowed broadcast of entity state PDUs and weapon state PDUs. The World State Manager also provided entity state information for networked entities. While the overall results were favorable, areas for improvement include a small portion of the cockpit modeling and a limited DIS interface. The results in each of the six areas of VC research (reconfigurable software architecture, reconfigurable cockpit geometry models, reconfigurable simulation components, replacement of objectsim functionality, improved cockpit interface, and distributed simulation interface) are covered in detail in the following sections.

Table 5-1. Reconfigurable Software Architecture's Requirements.

1. Reconfigurable Software Architecture	
1.1. Allow rapid reconfiguration of both aircraft geometry and simulation components (see REQs 2 and 3)	Architecture should allow switching between aircraft in less 1 second.
1.2. Increase flexibility of simulation framework by eliminating ObjectSim's constraints	Allow all Performer functionality to be available to developers by completely removing ObjectSim from VC
1.3. Provide architecture that will support all needed simulation components and be extensible	Utilize container-based approach to storing simulation data
1.4. Support Multiple Aircraft	Configuration supports at least two aircraft
1.4.1. Integrate existing F-15 VC into CODB architecture	CODB F-15 VC works identically to 1995 F-15 VC.
1.4.2. Develop a F-16 VC	F-16 VC with appropriate aircraft model, sensors and weapons (REQ 3).

5.1. Reconfigurable Computer Architecture

The result in the reconfigurable computer architecture area are promising. The CODB was a good replacement for ObjectSim in this application. The architecture also allowed for quick reconfiguration, less than 0.1 seconds. Figures 5-1 and 5-2 show a view of the aircraft and cockpit of the F-16. Figures 5-3 and 5-4 show the F-15 aircraft and cockpit after reconfiguration. Each requirement for

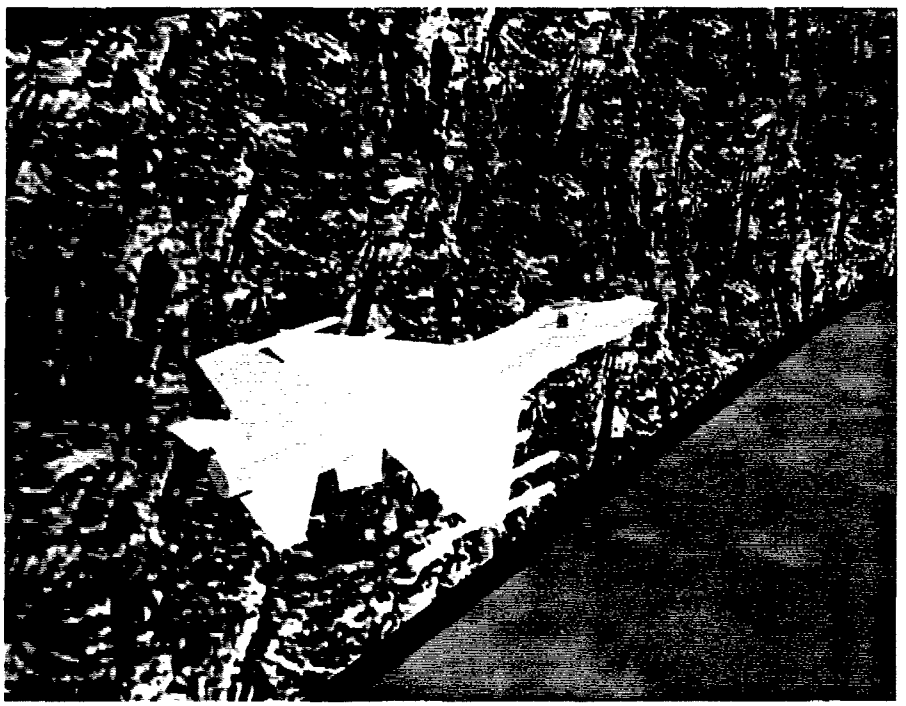

Figure 5-1. F-16 Aircraft.

Figure 5-2. F-16 Instrument Panel.

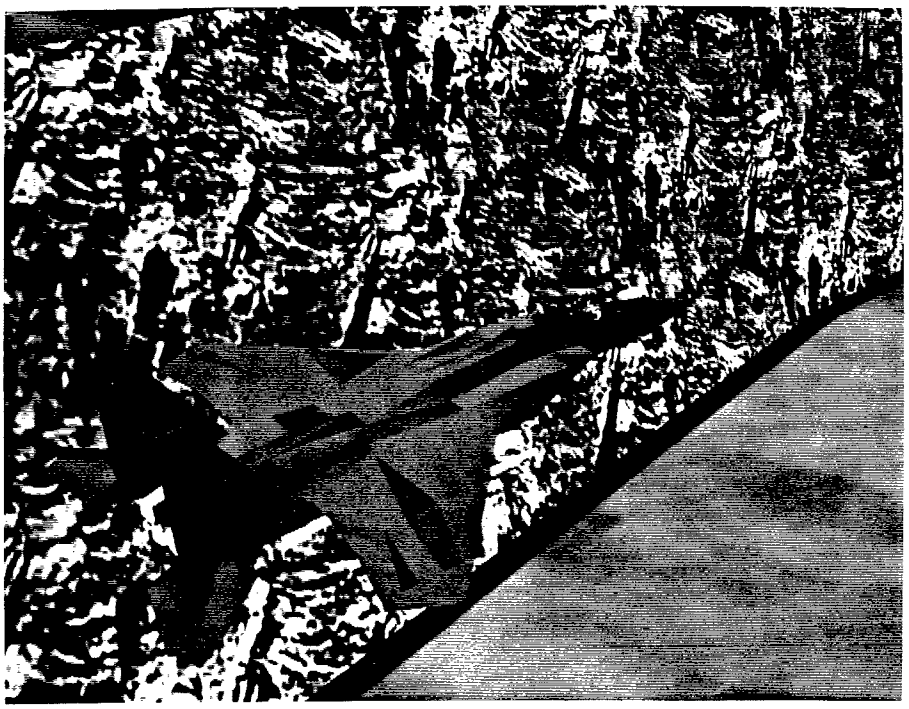

Figure 5-3. F-15 Aircraft.

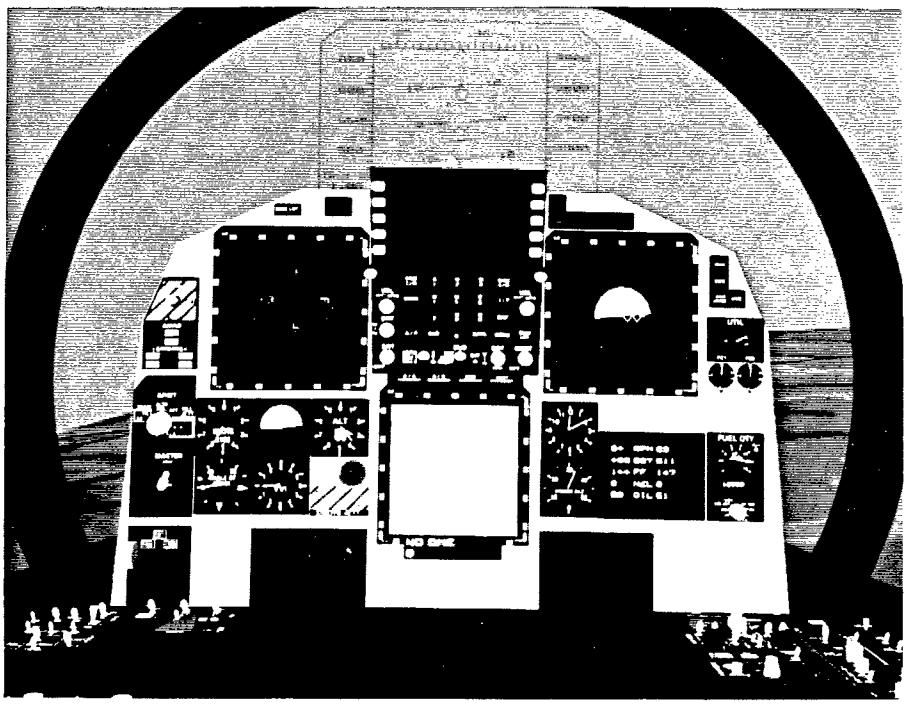

Figure 5-4. F-15 Instrument Panel.

this section will be discussed with its goal and the result actually achieved. For convenience Table 5-1 restates the requirements and goals for this section from Table 3-1.

5.1.1. Requirement 1.1: *Increase flexibility of simulation framework by eliminating ObjectSim's constraints.* All ObjectSim components were removed from the simulation easing the integration of I/O devices into the simulation. All Performer functionality was visible to the simulation developer.

5.1.2. Requirement 1.2: *Allow rapid reconfiguration of both aircraft geometry and simulation components.* The RRVC met its goal of switching between a F-15 aircraft and a F-16 aircraft in less than one second. In fact the switch was accomplished in one simulation frame, less than 0.1 seconds. Each component of the RRVC switched to the appropriate model for that aircraft type, including aerodynamics, sensors, weapons, aircraft geometry, cockpit geometry, and DIS representation. The switching of the aircraft geometry and cockpit geometry was visually observed. While a software debugger was used to determine if the other simulation components were correctly switched.

5.1.3. Requirement 1.3: *Provide architecture that will support all needed simulation components and be extensible.* The CODB approach was used extensively in the development of the RRVC. The CODB eased the development process by focusing on containers instead of classes. For instance while awaiting the completion of the World State Manager, testing of the DIS interface was needed; however, no functionality was complete in the WSM. For testing, a non-DIS WSM was developed that filled the WSMEntityStruct container in the CODB. All access to DIS information was then accomplished using this container. When the WSM became available, the non-DIS WSM was commented out and the container was then filled by the DIS-WSM without any impact to the development time. The container approach also aided reconfiguration because the many of the components did not care what type of aircraft was being simulated only what information was stored in the containers. For instance, the weapons do not need to know what aircraft they are attached to, they only need to know the position and orientation of the aircraft, provided in the CODB.

5.1.4. Requirement 1.4: *Support Multiple Aircraft.* The goals for this requirement was to support both the current F-15 VC and a F-16 VC. Both goals were successfully reached. The F-15 VC was integrated into the CODB architecture and into the Airplane class. All F-15 VC components were tested and provide the same functionality under the CODB architecture as they did under the ObjectSim architecture. A F-16 was created from scratch using the CODB and also integrated into the Airplane class. More information on the results of the F-16 development are included in discussion of requirements 2 and 3. The airplane class is used to provide a place for all aircraft simulation components and automates the switching between aircraft types. The Airplane class can easily be extended to include as many different aircraft types as needed.

Table 5-2. Reconfigurable Cockpit Geometry Requirements.

Requirement	Goal
2. Reconfigurable Cockpit Geometry Models	
2.1 Allow switching between different aircraft cockpits	Switch between F-15 and F-16 cockpits in less than one second
2.2. Create photo-realistic F-16 Instrument Panel	Cockpit instruments same size, shape, and coloring of actual display
2.2.1. Use anti-aliased textures for cockpit text	All text in cockpit is anti-aliased to increase realism
2.2.2. Design textures to be reusable in other aircraft	All text textures can have any foreground or background color and any font size
2.2.3. Create realistic dials and needles for instruments	Entire instrument panel implemented in DWB to allow addition of materials and lighting effects

5.2. Reconfigurable Cockpit Geometry

The reconfigurable cockpit geometry area met all requirements met, see Table 5-2. Photo-realistic displays were created for the F-16 cockpit. All instruments were correctly sized and positioned on the instrument panel. Material and coloring effects were duplicated as precisely as possible. Texture maps were used for the text in most of the cockpit instruments, except for the angle of attack indicator and vertical velocity indicator. All texture maps were created using the technique discussed in section 4.2. All dials and needles were implemented in DWB providing both material and lighting effects. F-16 pilots were used to evaluate the accuracy of the displays and found them to be very good [GREIR96], [ULST96]. Photos of the entire instrument panel along with individual instruments is provided in in Appendix A.

5.2.1. Requirement 2.1: *Allow switching between different aircraft cockpits.* The Performer tree implemented allows the application to switch aircraft geometries in less than 0.1 seconds, easily meeting its goal. Each future cockpit will be placed at the same level of the Performer tree and switching will take

place in the same manner. Future switching may not happen as quickly depending on the amount of system and texture memory required for future cockpits. If the available amount of memory is exceeded the operating system will swap out memory that will cause a delay in the switching. Currently, the F-15 and F-16 cockpits both easily fit into the computer's 190 megabytes of system memory and 4 megabytes of texture memory.

5.2.2. Requirement 2.2: *Create photo-realistic F-16 Instrument Panel.* The F-16 VC cockpit is the same size and dimensions when measured against an actual F-16 cockpit. All dimensions were measured down to 0.125 inches. Anti-aliased textures were used for most of the instruments giving an added realism beyond what was possible with the vector fonts available in GL (Requirement 2.2.1). The angle of attack (AOA) and vertical velocity (VVI) indicators' displays were updated using GL's vector font. Both displays in an actual F-16 are implemented mechanically by a sliding tape that could not be easily duplicated in DWB. The tape slides behind the instrument face to display the actual reading for the instrument. Creating a flat-tape is possible using DWB; however, the tape would be too long and would protrude into the displays of other instruments. For this reason, a GL font-based approach was used that simulates the sliding tape. All other instruments used text textures that were created using GIMP and can be reused for other aircraft types (Requirement 2.2.2). Finally, all dials and switches were implemented using DWB, except for the AOA and VVI (Requirement 2.2.3). The dials were a clear improvement over the dials in the F-15 VC in terms of appearance (through the use of anti-aliased textures) and functionality (because the dials actually turned instead of flashing the current value at discrete steps). When monitoring the F-16 altimeter dials, it was clearly evident with a quick glance how fast the altitude was changing from the motion of the dials. On the other hand, the F-15's altimeter's dials requires the pilot to actually read the values on the dial and consciously determine how quickly they are changing.

Table 5-3. Reconfigurable Simulation Components' Requirements.

Requirement	Goal
3. Reconfigurable Simulation Components	
3.1. Develop reconfigurable aircraft aerodynamic model	Aircraft aerodynamic model that can represent many types of aircraft
3.1.1. Model must be able to represent multiple aircraft based on data alone	Model will represent both the F-15 and F-16 aircraft and have data to support additional aircraft
3.1.2. Model must utilize CODB based input and output	All input and output from model is CODB based
3.2. Develop reconfigurable radar model	A simple CODB-based radar model that represent multiple aircrafts' field-of-views.
3.2.1. Radar must be able to change field-of-views while running	Any radar field of view attributes can be changed during execution of the application
3.1.2. Model must utilize CODB based input and output	All input and output from model is CODB based
3.3. Develop reconfigurable weapons controller	Weapons controller will support all current weapons and multiple aircraft
3.3.1. Modify existing weapons controller to support multiple aircraft	A single weapons controller that will support multiple aircraft
3.3.2. Provide CODB container for weapons status data	Weapon status information will be available to all components in CODB
3.3.3. Create new bomb that will be guided to target by a Virtual GPS receiver	Utilize existing bomb model to create an additional bomb type that will use GPS for guidance and hit target

5.3. Reconfigurable Simulation Components

The three primary reconfigurable simulation components, aerodynamics, weapons, and radar, met all the desired goals (see Table 5-3). All the models use CODB data for both input and output. The aerodynamics model is able to represent many different aircraft and currently has data on the F-15, F-16, F-18, F-5E, and the A-10. The radar model can represent any radar field of view and can be manipulated by the application based on the current aircraft type's radar field of view. The weapons controller has been modified to work for any number of aircraft and provide representative weapons loads for those aircraft. Finally, the bomb model was changed to create a GPS guided bomb. The bomb utilizes a virtual GPS receiver to hit its target. The reconfigurable simulation components work as intended.

5.3.1. Requirement 3.1: *Develop reconfigurable aircraft aerodynamic model.* The results of the reconfigurable aircraft model are based on its ability to reconfigure between different types of aircraft (Requirement 3.1.1) and its ability to utilize CODB input and output (Requirement 3.1.2). The model was not evaluated for performance accuracy because this model has been flown and accepted by Wright Laboratory's Flight Simulation Branch. The model was made to be reconfigurable and testing was accomplished on switching between F-15 and A-10 aircraft. The execution of the Airplane class was examined in a software debugger to ensure the correct model was being executed. This provided hard data

on the switching of models. The difference in models is quickly evident to the pilot when switching between an high-performance F-15 and a relatively slow A-10. All the performance the pilot has come to expect with the F-15 is suddenly lost with the A-10, including turn rate, top speed, and maximum g-forces. The HotasStruct container was used as input to control the model, with all buttons and switches acting as expected. Unfortunately, the throttle, stick, and rudder input device that is available in the Graphics Laboratory is in a F-15 configuration. All F-16 buttons and switches had to be given an F-15 counterpart to allow a pilot to perform operations with switches that are available in the F-16 and not available in the F-15.

5.3.2. Requirement 3.2: *Develop reconfigurable radar model.* The radar model met both of its requirements: dynamic field of views (Requirement 3.2.1) and utilization of CODB for input and output (Requirement 3.2.2). The model was made to change the radar viewpoint during the execution of the simulation. The radar updated only those targets in its new field of view. The CODB RadarStruct is used as an output container for the model. This container was used to designate radar targets for weapon targeting and to update the multi-function display's radar screen. For accuracy testing, targets were placed at known locations in the environment with a non-moving radar (aircraft was stationary). The radar screen provided visible proof that the radar model was working. Figure 5-5 shows a sample of the radar screen / display.

5.3.3. Requirement 3.3: *Develop reconfigurable weapons controller.* The weapons controller is used for both the F-15 and F-16 VCs and keeps weapon information for each type of aircraft (Requirement 3.3.1). The initial weapon load information is stored in a configuration file that is passed into the init method of the weapons controller class. The number of weapons displayed on the HUD is used to verify that the correct weapons load is available for each aircraft. In addition when reconfiguring the aircraft the F-16's MFD's weapons screen also reflected the current weapons load for that aircraft, along with the F-15's HUD display. The WeaponsStruct CODB container is used by the HUD and MFD to

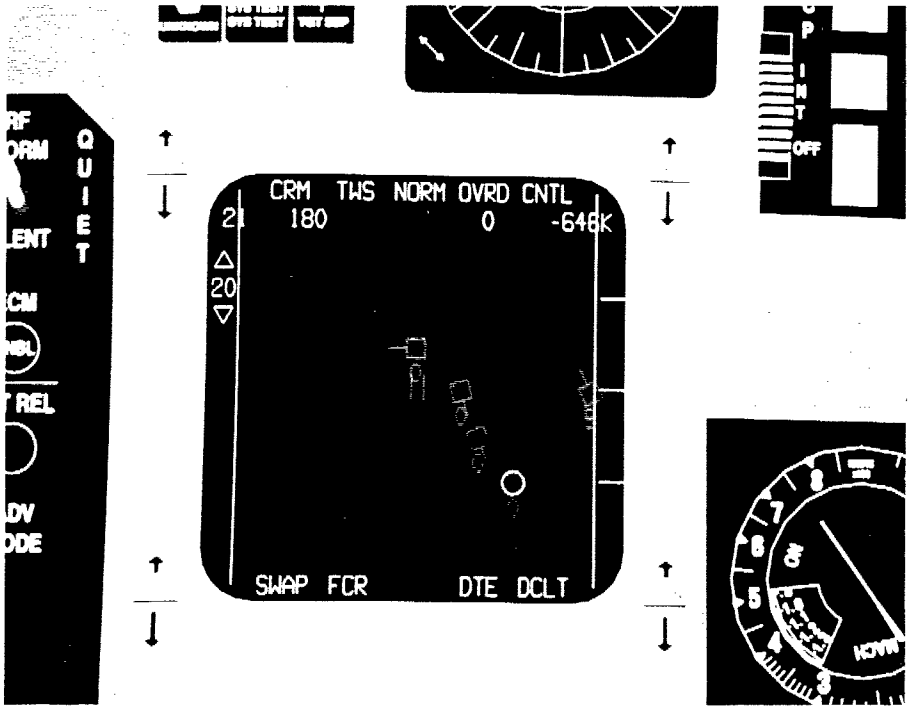

Figure 5-5. Sample Radar Display on Multi-Function Display

update their displays (Requirement 3.3.2). Weapons also utilize the CODB's HotasStruct for weapon release information.

5.3.4. Requirement 3.3.3: *Create new bomb that will be guided to target by a Virtual GPS receiver.*

The GPS-guided weapon works based on the bomb model that is part of the VC. The VC's bomb model uses perfect information to guide it to target and always hits the target. This model is greatly simplified; but, provides a starting point for modeling the interface for a GPS-guided munition. A more detailed bomb model must be implemented to realistically test the ability of the virtual GPS receiver to guide a bomb to target. Performance of the Virtual GPS receiver was consistent with the error rate expected in an actual GPS receiver. The virtual GPS receiver provided a new GPS position every simulation frame that was used for guidance. For information on the accuracy of the Virtual GPS Receiver see Captain Gary Williams' 1996 thesis [WILL96]. Given the errors associated with the GPS position the bomb had a miss distance of approximately 11.95 meters.

Table 5-4. Replacing ObjectSim Functionality Requirements.

Requirement	Goal
4. Replace ObjectSim Functionality	
4.1. Replace origin-centered viewpoint algorithm	Overcome Performer viewpoint resolution problem
4.2. Structure Performer tree to support weapons model and viewpoint algorithm	Create same top-level tree structure as that in 1995 ObjectSim VC

5.4. Replacing ObjectSim Functionality

The ObjectSim functionality was replaced meeting both requirements in this area (see Table 5-4). The adoption of the ObjectSim View algorithm and the its Performer tree eased the integration of the existing F-15 VC and the CODB architecture. ObjectSim also provides a DIS interface for network simulation. The replacement of the VC's DIS interface will be discussed in Section 5.6.

5.4.1. Requirement 4.1: *Replace origin-centered viewpoint algorithm.* The elimination of the viewpoint jittering was quickly apparent after implementing the viewpoint-at-origin algorithm in ObjectSim's View class [SNYD93]. No viewpoint jittering has been observed in the hundreds of trial runs made with the RRVC.

5.4.2. Requirement 4.2: *Structure Performer tree to support weapons model and viewpoint algorithm.* A Performer tree in the same basic structure as the ObjectSim tree. The results of achieving this requirement are observed in other areas, such as elimination of viewpoint jitter and correct weapons display. Viewpoint jitter is discussed above. The weapons are switched from the VC portion of the tree while attached to the aircraft and then moved to the DIS players portion of the tree when released. This switch is built into the model and the Performer tree had to maintain the same structure to allow all weapons to be displayed correctly.

Table 5-5. Replacing AFIT Pod Interface Requirements.

Requirement	Goal
5. Improved cockpit interface	
5.1. Allow selection of three dimensional panels and instruments	Point and click interface for any type of geometry in a single easy to use class
5.2. Allow selection of buttons from several different panels at once	Any button can be selected at any time.
5.3. Improve interface with switches and dials	Dials and switches move left / right and slow / fast as desired.

5.5. Replacing AFIT Pod Interface

The replacement of the AFIT Pod interface provided the greatest change between using the F-15 VC and the F-16 VC. The F-15 cockpit was not changed from the AFIT Pod interface to the new Selection Manager within the F-16 VC, it provides a good contrast between the two selection styles. The Selection Manager provides an interface to the buttons and switches that are more natural then before. The movement is more natural because it is more consistent with a human's natural interaction with switches and dials. Any type of button or switch can be selected by the user with a minimum of overhead or setup. Each of the requirements was designed, implemented and tested in the RRVC and met all desired goals (see Table 5-4).

5.5.1. Requirement 5.1: *Allow selection of three dimensional panels and instruments.* The user of the RRVC can select any shape of panel and is no longer limited to two dimensions.

5.5.2. Requirement 5.2: *Allow selection of buttons from several different panels at once.* The user can select any button or switch in the simulation. All buttons and panels are active all the time. No user interaction is required to make a panel active as is the case with the F-15 VC.

5.5.3. Requirement 5.3: *Improve interface with switches and dials.* The interface with switches and dials is greatly improved. The F-15 cockpit allows only a single mouse button to be used to select a switch or dial (Note: A switch has several discrete settings, while a dial has a continuos motion). Therefore, switches are implemented to move only in one direction and only change direction when the limit is reached. The Selection Manager in the F-16 cockpit allows the user to use the left or right mouse buttons to turn a switch in either direction. The left and right motion for dial is implemented in the same manner, with the middle mouse button used to increase the rate of motion.

Table 5-6. Distributed Simulation Interface Requirements.

Requirement	Goal
6. Distributed Simulation Interface	
6.1. Send and receive entity state information for aircraft	Communicate Entity State PDU information with DIS using CODB
6.2. Broadcast weapon state information on network	Communicate Fire, Entity State, and Detonate PDUs using CODB.
6.3. Display all network entities to pilot	Network entities appear correctly in Performer scene

5.6. Distributed Simulation Interface

The Distributed Simulation Interface met all of its requirements as defined in Table 5-6. All DIS communication is via the CODB using the World State Manager. Ownship information for aircraft and weapons state is broadcast over the DIS network . Network entity state information is received and displayed to pilot as part of the world scene. The results of the DIS interface were primarily results observed in the ModSAF application. ModSAF allows developers to create and view DIS exercises [LORA95]. RRVC broadcasts PDU information that is then viewed in ModSAF, for accuracy. The accuracy of entity state, weapon's fire, and weapon's detonate PDUs' performance can all be observed in this manner. In turn, entities were created in ModSAF that were then broadcast over the network. The RRVC out-the-window and radar displays are used in conjunction with the ModSAF displays to compare positions of the ModSAF entities with the RRVC aircraft. Figure 5-6 shows a ModSAF screen with four tanks. Figure 5.7 shows the RRVC's terrain. Using the same terrain in both ModSAF and the RRVC allowed comparison entity position based on terrain features such as roads and rivers. For instance, notice the hangman's noose right below the tanks on the ModSAF figure. This noose corresponds to the noose in the center of the RRVC terrain figure. Terrain features were a valuable tool in comparing DIS positional data and the RRVC's network entities' positional data. Testing in this area focused on determining the accuracy of the RRVC's position to other DIS entities and the DIS entities' position in the RRVC. Testing did not include the World State Manager's ability to send and receive PDUs, that was verified by the WSM developer. Round_Earth_Utils provides methods to convert between DIS round-earth coordinates and Performer flat-earth coordinates. The Round_Earth_Utils class could introduce a source of error to the process in the conversion process; however, this class has been verified as part of a previous thesis effort [ERIC93].

5.6.1. Requirement 6.1: *Send and receive entity state information for aircraft.* Entity state information is being accurately portrayed to both the network and the pilot in the RRVC based on observation from both viewpoints.

5.6.2. Requirement 6.2: *Broadcast weapon state information on network.* ModSAF shows weapon entity state information and detonation information. Both of these attributes were observed in

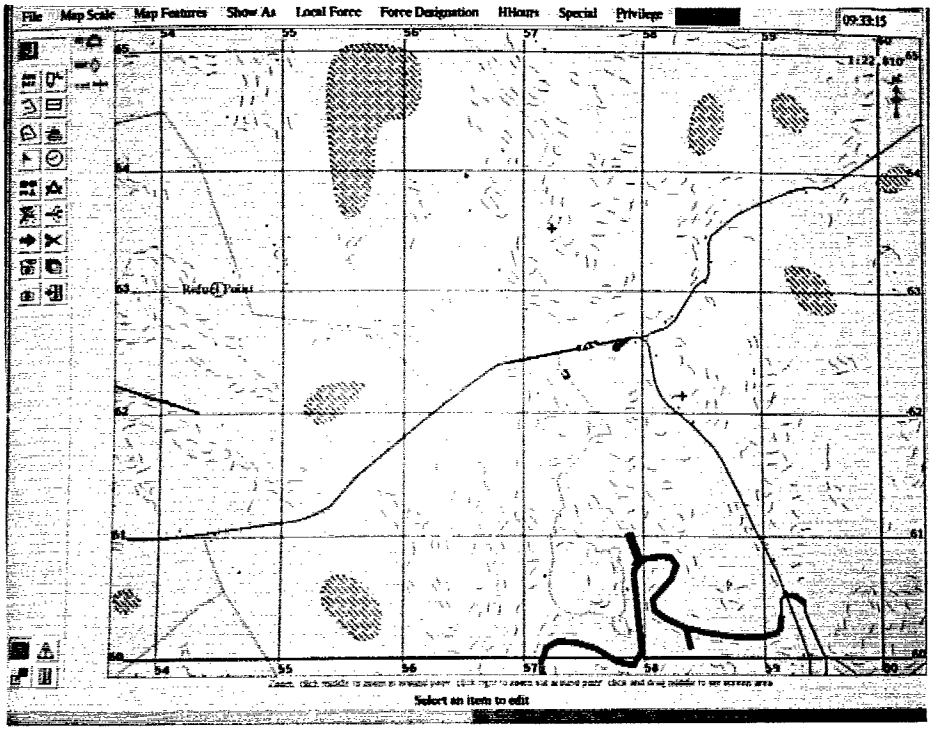

Figure 5-6. ModSAF Terrain with Four Tanks.

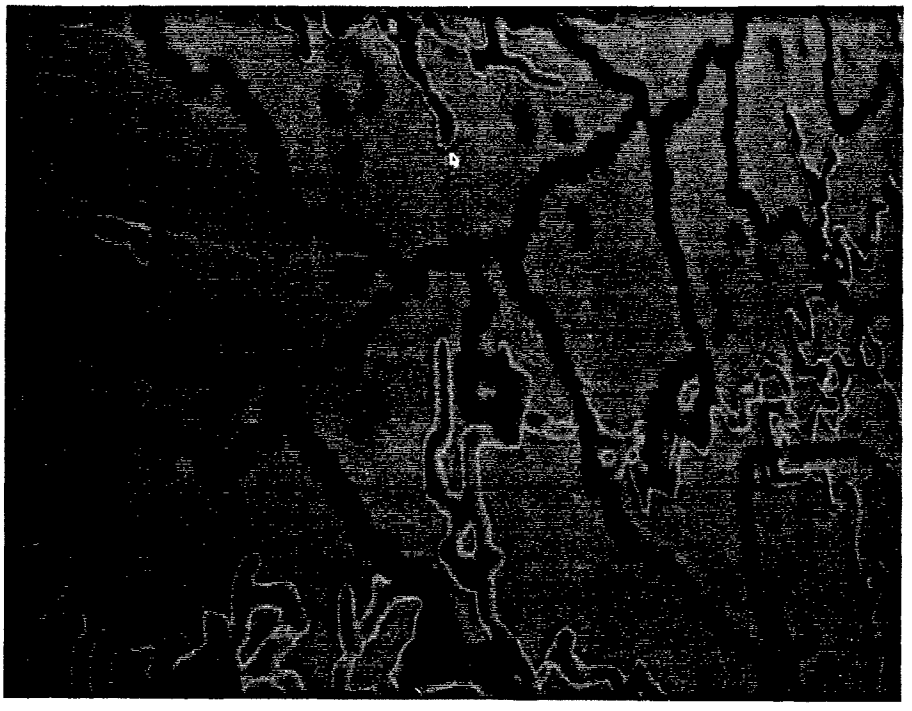

Figure 5-7. RRVC Fort Knox Terrain.

ModSAF, using the same procedures used for evaluating DIS entity positional data, and found to be accurate.

5.6.3. Requirement 6.3: *Display all network entities to pilot.* For testing purposes, ModSAF was used to define an exercise over Fort Knox, Tennessee. The Fort Knox terrain is loaded into the RRVC as its terrain file. Comparison of the ModSAF and RRVC target locations against the underlying terrain were made for accuracy. All four targets were located over identical terrain features in both ModSAF and the RRVC.

5.7. Conclusion

The RRVC met all the goals for the project and had good results in all areas. The RRVC reconfigured between a F-15 and F-16 in less than 0.1 seconds. All F-16 instruments are functional and are accurately sized, shaped, and oriented. The RRVC was flown by two F-16 pilots who commented favorably on both the aircraft model and the instruments. The weapons models and radar models performed as required; however, both models are low-fidelity and provide a starting place for future development. The Selection Manager provides a substantial increase in functionality over the AFIT Pod, at considerably less complexity. The WSM provides a DIS interface to provide the RRVC with network entity information and broadcast RRVC information to the network entities. The RRVC provides a framework for future reconfigurability development and a pilot training tool on the DIS network.

6. Conclusions and Future Work

Chapter 6 discusses conclusions I have made in developing a Rapidly Reconfigurable Virtual Cockpit and areas of future research for the RRVC. The RRVC research project demonstrates how a reconfigurable, distributed virtual cockpit environment can be assembled. The RRVC allows pilots to train in either an F-15 or F-16 cockpit under a single application. The architecture provides a framework for future VC research for additional aircraft types and more realistic models. This chapter discusses specific conclusions for the five main research areas of this thesis effort: reconfigurable computer architectures, reconfigurable cockpit geometry, reconfigurable simulation components, AFIT Pod replacement, and DIS interface. The RRVC also opens the door for many other areas of research and development including reconfigurable software models and virtual cockpit interfaces. The largest area for future development is DIS training environments. The RRVC is a training tool that can be used to train pilots across the country in combat tactics or mission rehearsal. The training can be conducted with other pilots in the same building or on the other side of the world. The RRVC is a low-cost training platform that prototypes many unique training opportunities for operational squadrons.

6.1. Accomplishments

The RRVC research builds upon the foundation laid by previous thesis students who include the following: Switzer, Gerhard, Erichson, McCarty, Diaz, Kesterman, Snyder, and Sheasby. These students laid the groundwork for a virtual cockpit in the areas of modeling and distributed simulation in the AFIT Graphics Laboratory. Their research allowed the development of a photo-realistic virtual F-15 cockpit. My research builds the concept of a F-15 Virtual Cockpit into an application that can support multiple aircraft in a single DIS exercise. The development of a reconfigurable architecture utilizing the CODB is of equal importance. The advisor for this research, Lt Col Martin Stytz, developed the concept of the CODB [STYT97]. The CODB is a simulation architecture built upon containers [STYT97]. The RRVC research is one of the initial users of the CODB architecture and the only project to employ the CODB in a reconfigurable application.

The specific accomplishments of this research are that the project has met all the requirements and goals discussed in Chapter 3. The RRVC shows that a single application can support multiple VC aircraft types. Additionally, the research project expands the number of CODB simulation components that are available for other simulation developers. The CODB-based reconfigurable aircraft model developed during this research is used by two other thesis students developing an Artificially Intelligent Wingman [BENS96][ZURI96]. The model allowed them to quickly create and control an F-16 aircraft using the CODB. This research effort also enhanced the Selection Manager class that is being used by Wells [WELL96]. The RRVC research has not only built upon previous VC efforts, but has laid the groundwork for future VC research in the areas of reconfigurable simulation components and distributed training exercises.

6.2. Conclusions

6.2.1. Reconfigurable Computer Architecture. The reconfigurable computer architecture performs as intended. The selection of the CODB as the framework for simulation components greatly simplified the components' interfaces. The architecture is extensible and will allow future developers to easily integrate additional aircraft types into the framework. Removing ObjectSim eliminated the learning curve associated with ObjectSim and increased developer flexibility. Keeping ObjectSim would have reduced the amount of time needed to develop the RRVC application because of ObjectSim's built in DIS. In addition, no ObjectSim functionality would have had to been replaced. However, the amount gained by replacing ObjectSim in increased flexibility and decreased complexity outweighed any advantages to keeping ObjectSim architecture.

The CODB was the most important choice in the architecture's development. It not only eased the component interface task also eased integration. When the DIS component was unavailable for integration and testing, a simple DIS simulator was developed that wrote representative DIS entity data values to the DIS CODB container. Testing of components that required DIS information was able to continue on schedule. When, the DIS component was available the DIS simulator was removed allowing the DIS

component to fill the CODB container. The RRVC worked the same with the DIS component as it had with the simulator and because it only needed to read the CODB container no input/output methods had to be modified or developed.

6.2.2. Reconfigurable Cockpit Geometry. The development of Cockpit Geometry is the most time consuming portion of developing a Virtual Cockpit. The slight differences in each of the instruments make reuse difficult. However, one area for reuse exploited in the F-16 VC design is the reuse of textures. While the airspeed indicator may be different in a F-15 and F-16, it has many of the same words on the instrument providing an opportunity for reuse. The design of reusable textures did not save much texture memory in this situation because the F-15 VC cockpit textures were already created but it should save memory in future aircraft implementations.

DWB was a satisfactory tool in which to develop the F-16's instrument panel. Problems such as small application errors and several crashes caused a lack of trust with the DWB application. DWB also provides almost no on-line help and the manuals are little more than a print out of man pages. The lack of help prevented me from discovering several DWB features that were not easily accessed through DWB's interface. An alternative to DWB is to build the instruments using GL. All the DWB models are eventually translated into Performer nodes and Performer is built upon GL, so using GL would eliminate the intermediate processing. However, developers may find GL modeling very complicated. I would recommend using GL only if the developer has a great deal of experience in developing models using GL. GL provides a great deal of flexibility; but, is many times more complicated then the menu-driven interface provided by DWB. In conclusion, DWB is an acceptable modeling tool for those who do not know GL and need to develop complicated models.

6.2.3. Reconfigurable Simulation Components. Reconfigurable simulation components greatly eased the development of the cockpit. I would recommend that reconfigurable components be used where ever possible. The models are difficult to develop; but, can sometimes be obtained from others involved in aircraft simulation (system program offices, labs, simulation organizations). The reconfigurable models are usually not perfect because they represent many different types of components. However, for the RRVC the fidelity provided by the reconfigurable aircraft model was more than acceptable. Several F-16 pilots

commented on the accuracy and performance of the reconfigurable aircraft model. The radar model developed for the F-16 VC provided basic functionality and is another component that should be made more realistic and reconfigurable. Most radars work in the same way and only have different powers, wavelengths, or search patterns. A radar model could be made reconfigurable in the same way as the aircraft model, by allowing this data to be determined based on radar type. Reconfigurable models give the developer a step up in developing a new aircraft simulation; however, they are not always the best choice. The instrument panels in aircraft are different enough that they should not be reconfigurable. Both the geometry and functionality of the cockpit are so different that each aircraft uses a separate model. Placing multiple instrument panels in a single model or class only increases the complexity of the class and does not ease the development process. In conclusion, reconfigurable components were a good fit with the RRVC; but, do not provide the answer in all situations (i.e., instrument panels).

6.2.4. AFIT Pod Replacement. The replacement of the AFIT Pod was the greatest single factor in decreasing the complexity of the RRVC. As stated in Chapter Four, the AFIT Pod requires a great deal of overhead for any selectable item. The AFIT Pod also limits where the items can be placed and how they can be selected. The Selection Manager allows the developer a great deal more freedom in employing user interaction with geometry in Performer. The interface also increases the speed in which dials and switches can be manipulated. Speed is important because pilots control the aircraft with the stick and throttle and do not like to spend a lot of time messing with switches and dials. On the other hand, an F-16 pilot told me that most of the dials and switches are setup pre-mission and are not manipulated during the mission. The Selection Manager also reduces the complexity of instrument panel. The F-15 VC has four different AFIT Pod classes that it utilizes to allow a pilot to select a button. The Selection Manager in the F-16 VC only uses one class and has more functionality. The Selection Manager provides an easy to use class with a great deal of flexibility for interface design.

6.2.5. DIS Interface. The World State Manager is a good choice for a basic DIS Interface. The WSM's development was a long process with a new release every week or two. As mentioned in the Reconfigurable Architecture, the lack of a DIS interface was overcome in certain situations. When a CODB structure for the interface existed, the development of a test driver to fill the CODB structure was a simple

task. However, in certain situations such as weapons fire and detonation, the CODB structure was not defined so work was delayed until that functionality became available in the WSM. Having Mr. Sheasby, the developer of the WSM, on-site was very important in flushing out the interface. The RRVC's DIS interface should be expanded to include cockpit PDU's that would allow one RRVC to have the same cockpit readings as another RRVC. This capability would allow a supervisor or trainer to monitor the status of a pilot flying a RRVC somewhere else on the network. To develop this new capability it is necessary to change the DIS interface. Without the developer on-site it would be extremely difficult to change the WSM. Using a commercial DIS interface for this requirement may have prevented the development of custom PDU's. However, a commercial DIS interface would have all basic DIS functionality from the start and the RRVC would not have had to develop any temporary DIS simulators.

6.3. Future Work

6.2.1. Reconfigurable Computer Architecture. The only future work projected to the reconfigurable computer architecture is a possible CODB improvement. Work is being done to move the CODB from a multi-process architecture to a multi-program architecture. As a multi-program architecture, the CODB could store data from multiple programs. The RRVC could use separate programs for input devices. Each input device could be started independently of the RRVC and remove a level of complexity from the application.

6.2.2. Reconfigurable Cockpit Geometry. The development of cockpit geometry is a time-consuming process; however, a time savings is possible by automating the development of textures. Currently, textures of words are created and each word is placed on a separate polygon. A better approach would be to have a class of textured letters. The class would take strings as input and use the textured letters to create the strings. A pfGroup representing the word would be returned, a Performer subtree consisting of several letter-textured polygons.

6.2.3. Reconfigurable Simulation Components. The development of separate RRVC aircraft could be greatly increased by utilizing more reconfigurable simulation components. Common aircraft components (radar, IR, counter-measures,...) should be identified and reconfigurable models developed to

emulate their functionality. The reconfigurable components must be realistic in each of their configurations. The weapon models and radar models in the RRVC have limited fidelity that should be improved before any pilot training is attempted.

6.2.4. AFIT Pod Replacement. Future work in this area is to explore how to best utilize the new Selection Manager. The Selection Manager should be integrated into the F-15 VC to provide a standard interface between cockpits. A great deal of code must be changed to accomplish this; but, it would decrease the complexity of the F-15 VC's cockpit.

6.2.5. DIS Interface. The DIS interface offers the greatest opportunity for future work. The RRVC is intended to be a single entity training device (one pilot), although they can communicate with other entities via DIS. The DIS interface allows pilots all over the world to fly together in a virtual environment. The RRVC should be expanded to allow a single aircraft to represent multiple aircraft entities in an exercise at the same time. The RRVC now switches between aircraft, so that only one aircraft exists at a time. However, by allowing a single pilot to control multiple aircraft, much larger mission scenarios could take place with fewer pilots. For instance, in a bombing mission rehearsal one pilot could use the RRVC to represent all the support aircraft (a fighter lead, refueling aircraft, jammers) in the simulation. The primary aircraft in the simulation, the bomber, would be flown by another pilot to carry out the primary mission. All the necessary aircraft types would be in the simulation and only use two pilots. Some controls would need to be added to the RRVC to allow each support aircraft to continue flying even though the support pilot may be flying a different aircraft at the time. A more typical approach would have a single RRVC represent all the support aircraft individually, one at a time. However, this would reduce the number of overall players in the simulation, affecting the accuracy of the mission's environment. Either method reduces the number of pilots needed to represent aircraft in the training exercise.

Another useful enhancement to the VC would be the addition of Cockpit PDUs. Cockpit PDUs would broadcast cockpit settings over the network. Other RRVCs could then be made to read the PDU and make their cockpit mirror another cockpit on the network. This would be especially valuable for training purposes. An instructor could monitor the progress of a student performing a tactic elsewhere in the country or the world. The instructor would not have to guess about what actions the pilot is taking because

the instructor can monitor all actions by looking at the cockpit. The instructor can monitor a trainee without being in the aircraft or peering over their shoulder. The instructor could also switch between all the players in a training exercise to evaluate multiple participants at a single workstation. The RRVC opens up the door for many different training scenarios and evaluation capabilities.

6.4. Conclusion

The RRVC project has fulfilled all of the requirements and goals of this thesis effort. The RRVC can switch between and F-15 and a F-16 in less than 0.1 second. The RRVC is built upon a reconfigurable architecture that provides a framework for future aircraft development. The CODB was a good choice for a RRVC architecture and eliminated many of the data storage issues that had to be addressed by previous VC developers. The WSM provides a good basic DIS interface for all CODB applications. However, there is room for improvement. More work needs to be accomplished in the area of reconfigurable simulation components. Expansion of the DIS interface would open the door for many unique, networked, aircraft training opportunities. The RRVC combines research in the areas of software architectures, reconfigurability, cockpit modeling, and virtual reality to create a unique, distributed, virtual, training device.

6. Conclusions and Future Work

Chapter 6 discusses conclusions I have made in developing a Rapidly Reconfigurable Virtual Cockpit and areas of future research for the RRVC. The RRVC research project demonstrates how a reconfigurable, distributed virtual cockpit environment can be assembled. The RRVC allows pilots to train in either an F-15 or F-16 cockpit under a single application. The architecture provides a framework for future VC research for additional aircraft types and more realistic models. This chapter discusses specific conclusions for the five main research areas of this thesis effort: reconfigurable computer architectures, reconfigurable cockpit geometry, reconfigurable simulation components, AFIT Pod replacement, and DIS interface. The RRVC also opens the door for many other areas of research and development including reconfigurable software models and virtual cockpit interfaces. The largest area for future development is DIS training environments. The RRVC is a training tool that can be used to train pilots across the country in combat tactics or mission rehearsal. The training can be conducted with other pilots in the same building or on the other side of the world. The RRVC is a low-cost training platform that prototypes many unique training opportunities for operational squadrons.

6.1. Accomplishments

The RRVC research builds upon the foundation laid by previous thesis students who include the following: Switzer, Gerhard, Erichson, McCarty, Diaz, Kesterman, Snyder, and Sheasby. These students laid the groundwork for a virtual cockpit in the areas of modeling and distributed simulation in the AFIT Graphics Laboratory. Their research allowed the development of a photo-realistic virtual F-15 cockpit. My research builds the concept of a F-15 Virtual Cockpit into an application that can support multiple aircraft in a single DIS exercise. The development of a reconfigurable architecture utilizing the CODB is of equal importance. The advisor for this research, Lt Col Martin Stytz, developed the concept of the CODB [STYT97]. The CODB is a simulation architecture built upon containers [STYT97]. The RRVC research is one of the initial users of the CODB architecture and the only project to employ the CODB in a reconfigurable application.

The specific accomplishments of this research are that the project has met all the requirements and goals discussed in Chapter 3. The RRVC shows that a single application can support multiple VC aircraft types. Additionally, the research project expands the number of CODB simulation components that are available for other simulation developers. The CODB-based reconfigurable aircraft model developed during this research is used by two other thesis students developing an Artificially Intelligent Wingman [BENS96][ZURI96]. The model allowed them to quickly create and control an F-16 aircraft using the CODB. This research effort also enhanced the Selection Manager class that is being used by Wells [WELL96]. The RRVC research has not only built upon previous VC efforts, but has laid the groundwork for future VC research in the areas of reconfigurable simulation components and distributed training exercises.

6.2. Conclusions

6.2.1. Reconfigurable Computer Architecture. The reconfigurable computer architecture performs as intended. The selection of the CODB as the framework for simulation components greatly simplified the components' interfaces. The architecture is extensible and will allow future developers to easily integrate additional aircraft types into the framework. Removing ObjectSim eliminated the learning curve associated with ObjectSim and increased developer flexibility. Keeping ObjectSim would have reduced the amount of time needed to develop the RRVC application because of ObjectSim's built in DIS. In addition, no ObjectSim functionality would have had to been replaced. However, the amount gained by replacing ObjectSim in increased flexibility and decreased complexity outweighed any advantages to keeping ObjectSim architecture.

The CODB was the most important choice in the architecture's development. It not only eased the component interface task also eased integration. When the DIS component was unavailable for integration and testing, a simple DIS simulator was developed that wrote representative DIS entity data values to the DIS CODB container. Testing of components that required DIS information was able to continue on schedule. When, the DIS component was available the DIS simulator was removed allowing the DIS

component to fill the CODB container. The RRVC worked the same with the DIS component as it had with the simulator and because it only needed to read the CODB container no input/output methods had to be modified or developed.

6.2.2. Reconfigurable Cockpit Geometry. The development of Cockpit Geometry is the most time consuming portion of developing a Virtual Cockpit. The slight differences in each of the instruments make reuse difficult. However, one area for reuse exploited in the F-16 VC design is the reuse of textures. While the airspeed indicator may be different in a F-15 and F-16, it has many of the same words on the instrument providing an opportunity for reuse. The design of reusable textures did not save much texture memory in this situation because the F-15 VC cockpit textures were already created, but it should save memory in future aircraft implementations.

DWB was a satisfactory tool in which to develop the F-16's instrument panel. Problems such as small application errors and several crashes caused a lack of trust with the DWB application. DWB also provides almost no on-line help and the manuals are little more than a print out of man pages. The lack of help prevented discovery of several DWB features that were not easily accessed through DWB's interface. An alternative to DWB is to build the instruments using GL. All the DWB models are eventually translated into Performer nodes and Performer is built upon GL, so using GL would eliminate the intermediate processing. However, developers may find GL modeling very complicated. I would recommend using GL only if the developer has a great deal of experience in developing models using GL. GL provides a great deal of flexibility; however, is many times more complicated then the menu-driven interface provided by DWB. In conclusion, DWB is an acceptable modeling tool for those who do not know GL and need to develop complicated models.

6.2.3. Reconfigurable Simulation Components. Reconfigurable simulation components greatly eased the development of the cockpit and should be used where ever possible. The models are difficult to develop, but can sometimes be obtained from others involved in aircraft simulation (system program offices, labs, simulation organizations). The reconfigurable models are usually not perfect because they represent many different types of components. However, for the RRVC the fidelity provided by the reconfigurable aircraft model was more than acceptable. Several F-16 pilots commented on the accuracy

and performance of the reconfigurable aircraft model. The radar model developed for the F-16 VC provided basic functionality and is another component that should be made more realistic and reconfigurable. Most radars work in the same way and only have different powers, wavelengths, or search patterns. A radar model could be made reconfigurable in the same way as the aircraft model, by allowing this data to be determined based on radar type. Reconfigurable models give the developer a step up in developing a new aircraft simulation; however, they are not always the best choice. The instrument panels in aircraft are different enough that they should not be reconfigurable. Both the geometry and functionality of the cockpit are so different that each aircraft uses a separate model. Placing multiple instrument panels in a single model or class only increases the complexity of the class and does not ease the development process. In conclusion, reconfigurable components were a good fit with the RRVC, but do not provide the answer in all situations (i.e., instrument panels).

6.2.4. AFIT Pod Replacement. The replacement of the AFIT Pod was the greatest single factor in decreasing the complexity of the RRVC. As stated in Chapter 4, the AFIT Pod requires a great deal of overhead for any selectable item. The AFIT Pod also limits where the items can be placed and how they can be selected. The Selection Manager allows the developer a great deal more freedom in employing user interaction with geometry in Performer. The interface also increases the speed in which dials and switches can be manipulated. Speed is important because pilots control the aircraft with the stick and throttle and do not like to spend a lot of time messing with switches and dials. On the other hand, an F-16 pilot commented that most of the dials and switches are setup pre-mission and are not manipulated during the mission. The Selection Manager also reduces the complexity of instrument panel. The F-15 VC has four different AFIT Pod classes that it utilizes to allow a pilot to select a button. The Selection Manager in the F-16 VC only uses one class and has more functionality. The Selection Manager provides an easy to use class with a great deal of flexibility for interface design.

6.2.5. DIS Interface. The World State Manager provided the basic DIS Interface; however, WSM's development was a long process with a new release every week or two. As mentioned in the Reconfigurable Architecture, the lack of a DIS interface was overcome in certain situations. When a CODB structure for the interface existed, the development of a test driver to fill the CODB structure was a simple

task. However, in certain situations such as weapons fire and detonation, the CODB structure was not defined so work was delayed until that functionality became available in the WSM. The RRVC's DIS interface should be expanded to include cockpit PDU's that would allow one RRVC to have the same cockpit readings as another RRVC. This capability would allow a supervisor or trainer to monitor the status of a pilot flying a RRVC somewhere else on the network. To develop this new capability it is necessary to change the DIS interface. Using a commercial DIS interface for this requirement may have prevented the development of custom PDU's. However, a commercial DIS interface would have all basic DIS functionality from the start and the RRVC would not have had to develop any temporary DIS simulators.

6.3. Future Work

6.2.1. Reconfigurable Computer Architecture. The future work projected to the reconfigurable computer architecture include a possible CODB improvement. Work is being done to move the CODB from a multi-process architecture to a multi-program architecture. As a multi-program architecture, the CODB could store data from multiple programs. The RRVC could use separate programs for input devices. Each input device could be started independently of the RRVC and remove a level of complexity from the application.

6.2.2. Reconfigurable Cockpit Geometry. The development of cockpit geometry is a time-consuming process; however, a time savings is possible by automating the development of textures. Currently, textures of words are created and each word is placed on a separate polygon. A better approach would be to have a class of textured letters. The class would take strings as input and use the textured letters to create the strings. A pfGroup representing the word would be returned, a Performer subtree consisting of several letter-textured polygons.

6.2.3. Reconfigurable Simulation Components. The development of separate RRVC aircraft could be greatly increased by utilizing more reconfigurable simulation components. Common aircraft components (radar, IR, counter-measures, etc.) should be identified and reconfigurable models developed to emulate their functionality. The reconfigurable components must be realistic in each of their configurations.

The weapon models and radar models in the RRVC have limited fidelity that should be improved before any pilot training is attempted.

6.2.4. AFIT Pod Replacement. Future work in this area is to explore how to best utilize the new Selection Manager. The Selection Manager should be integrated into the F-15 VC to provide a standard interface between cockpits. A great deal of code must be changed to accomplish this, but it would decrease the complexity of the F-15 VC's cockpit.

6.2.5. DIS Interface. The DIS interface offers the greatest opportunity for future work. The RRVC is intended to be a single entity training device (one pilot), although they can communicate with other entities via DIS. The DIS interface allows pilots all over the world to fly together in a virtual environment. The RRVC should be expanded to allow a single aircraft to represent multiple aircraft entities in an exercise at the same time. The RRVC now switches between aircraft, so that only one aircraft exists at a time. However, by allowing a single pilot to control multiple aircraft, much larger mission scenarios could take place with fewer pilots. For instance, in a bombing mission rehearsal one pilot could use the RRVC to represent all the support aircraft (a fighter lead, refueling aircraft, jammers) in the simulation. The primary aircraft in the simulation, the bomber, would be flown by another pilot to carry out the primary mission. All the necessary aircraft types would be in the simulation and only use two pilots. Some controls would need to be added to the RRVC to allow each support aircraft to continue flying even though the support pilot may be flying a different aircraft at the time. A more typical approach would have a single RRVC represent all the support aircraft individually, one at a time. However, this would reduce the number of overall players in the simulation, affecting the accuracy of the mission's environment. Either method reduces the number of pilots needed to represent aircraft in the training exercise. A similar configuration could be used by UAV controllers to control multiple UAVs from a single control station.

Another useful enhancement to the VC would be the addition of Cockpit PDUs. Cockpit PDUs would broadcast cockpit settings over the network. Other RRVCs could then be made to read the PDU and make their cockpit mirror another cockpit on the network. This would be especially valuable for training purposes. An instructor could monitor the progress of a student performing a tactic elsewhere in the country or the world. The instructor would not have to guess about what actions the pilot is taking because

the instructor can monitor all actions by looking at the cockpit. The instructor can monitor a trainee without being in the aircraft or peering over their shoulder. The instructor could also switch between all the players in a training exercise to evaluate multiple participants at a single workstation. The RRVC opens up the door for many different training scenarios and evaluation capabilities. The RRVC also provides an excellent testbed for future AFIT piloted simulation efforts.

6.4. Conclusion

The RRVC project has fulfilled all of the requirements and goals of this thesis effort. The RRVC can switch between and F-15 and a F-16 in less than 0.1 second. The RRVC is built upon a reconfigurable architecture that provides a framework for future aircraft development. The CODB was a good choice for a RRVC architecture and eliminated many of the data storage issues that had to be addressed by previous VC developers. The WSM provides a good basic DIS interface for all CODB applications. However, there is room for improvement. More work needs to be accomplished in the area of reconfigurable simulation components. Expansion of the DIS interface would open the door for many unique, networked, aircraft training opportunities. The RRVC combines research in the areas of software architectures, reconfigurability, cockpit modeling, and virtual reality to create a unique, distributed, virtual, training device.

Appendix A: F-16 Cockpit Photographs

This appendix contains pictures of the cockpit developed as part of this thesis. For an overall comparison, Figure A-1 provides an overall diagram of the F-16 cockpit. Figure A-2 shows the corresponding F-16 Virtual Cockpit. The remaining photographs provide close-ups of the cockpit and shows details not obvious from Figure A-2.

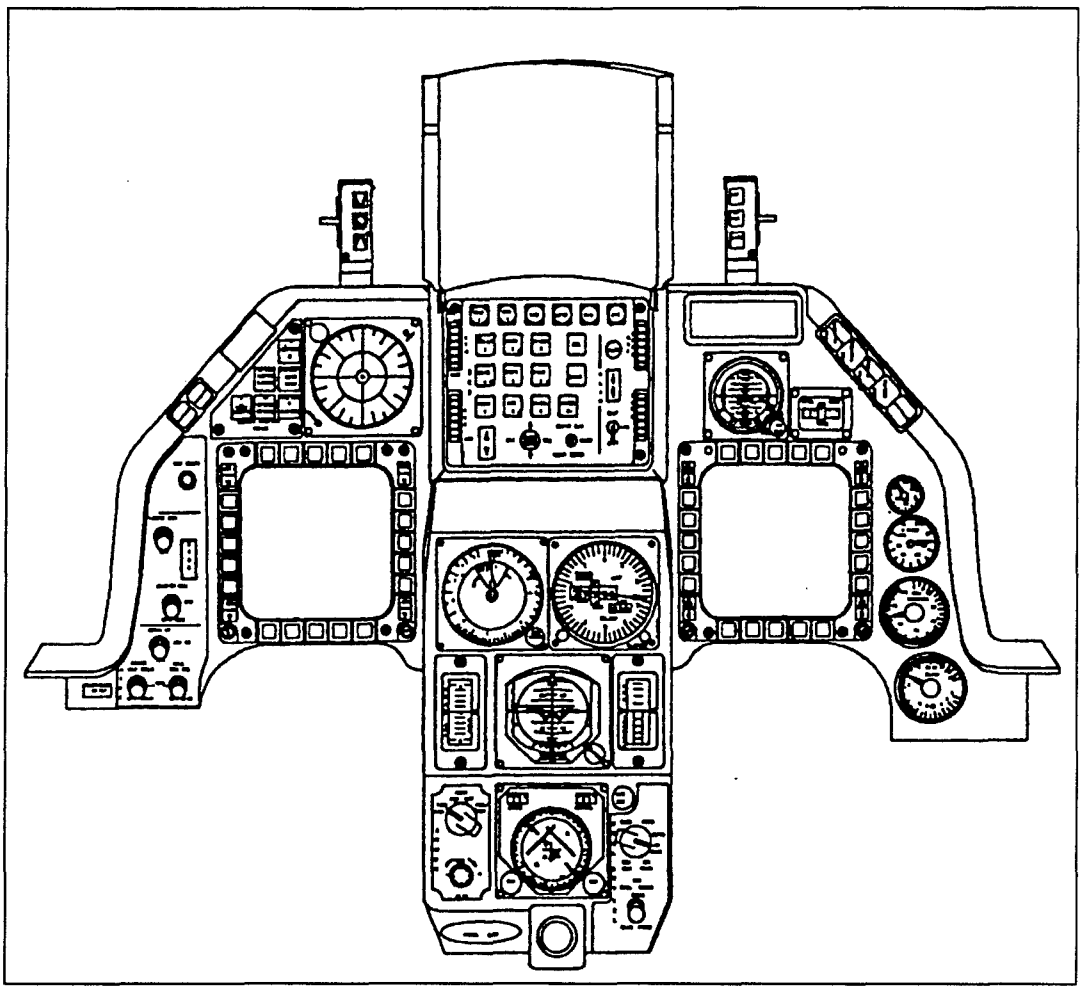

Figure A-1. F-16 Cockpit Diagram [OGDE94]

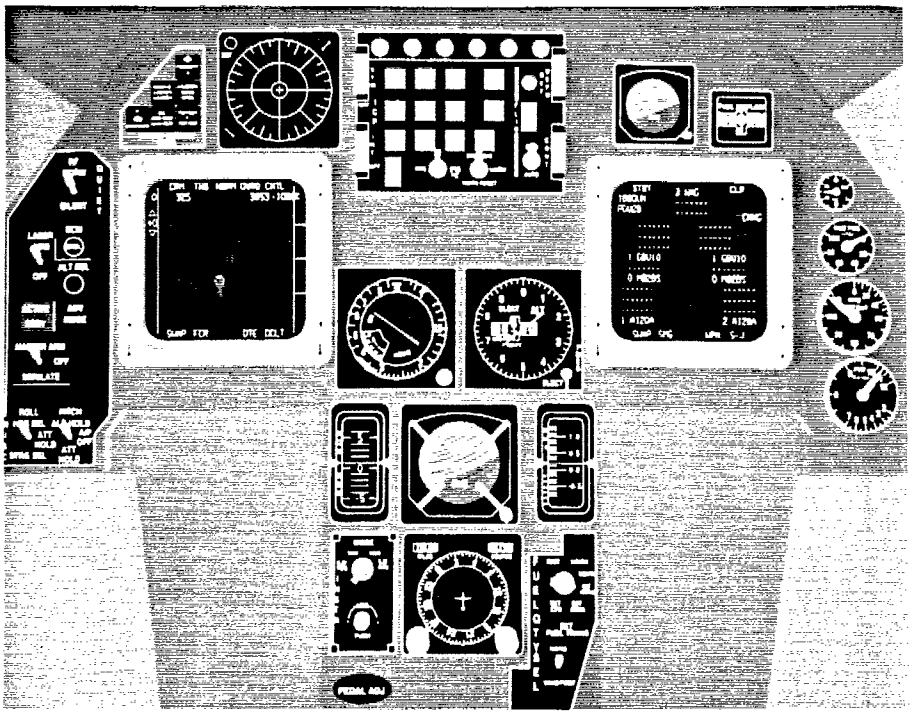

Figure A-2. Entire F-16 Virtual Cockpit Instrument Panel

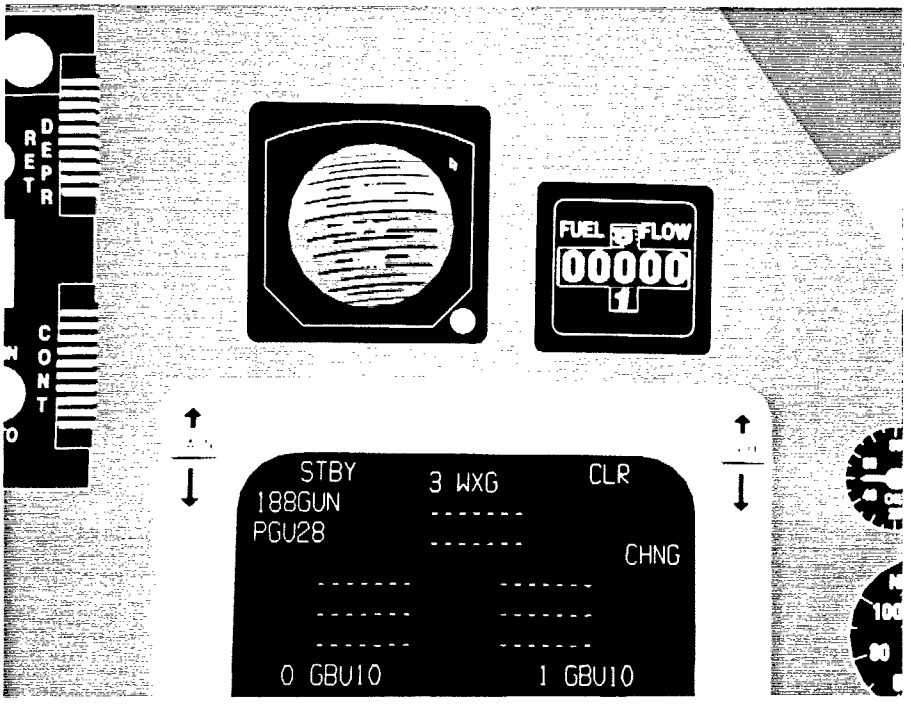

Figure A-3. Top Right Portion of F-16 Virtual Cockpit

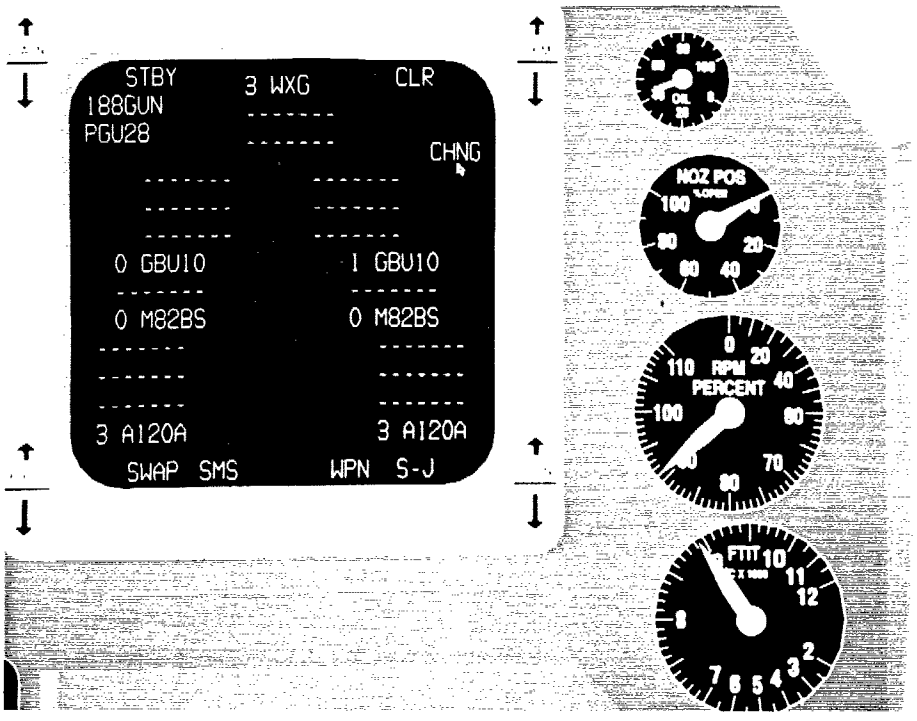

Figure A-4. Bottom Right Portion of F-16 Virtual Cockpit

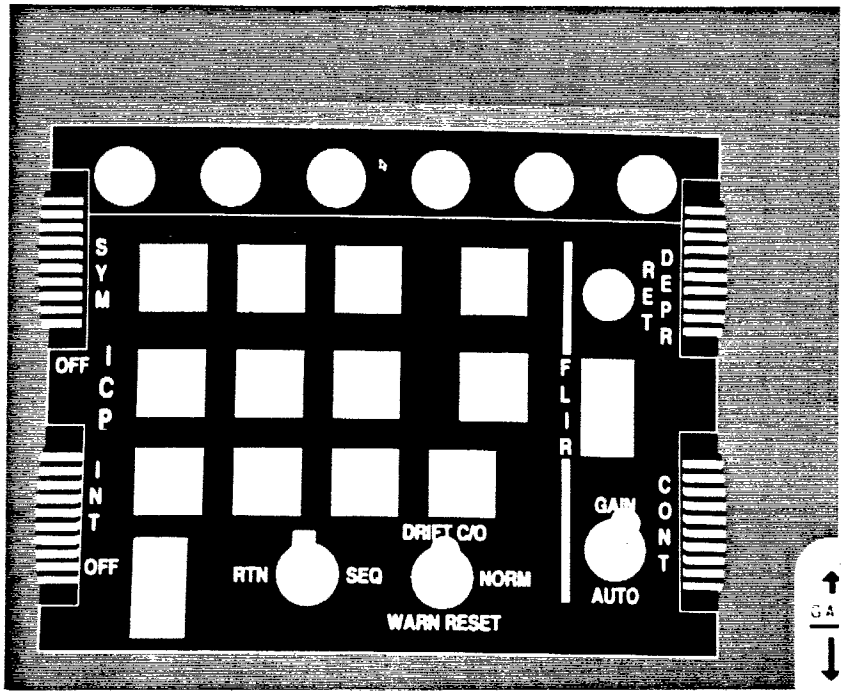

Figure A-5. Top Center Portion of F-16 Virtual Cockpit

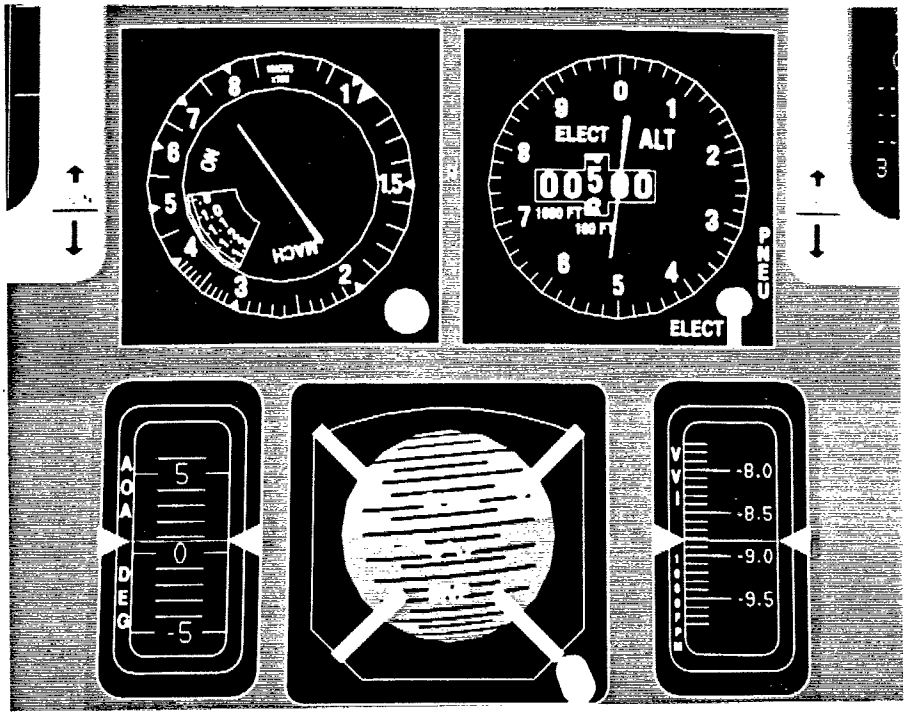

Figure A-6. Middle Center Portion of F-16 Virtual Cockpit

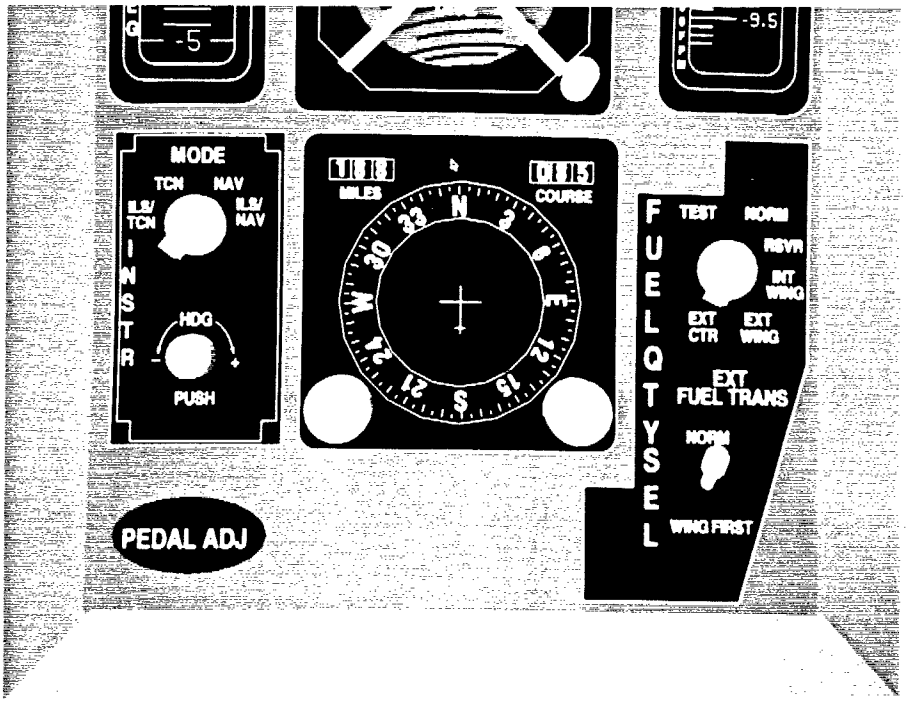

Figure A-7. Bottom Center Portion of F-16 Virtual Cockpit

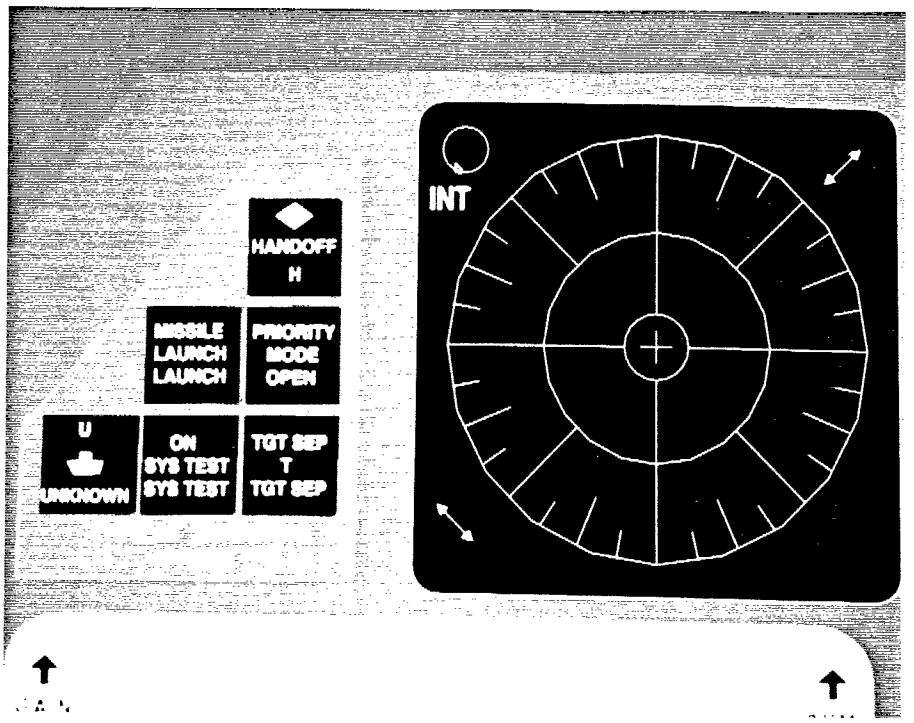

Figure A-8. Top Left Portion of F-16 Virtual Cockpit

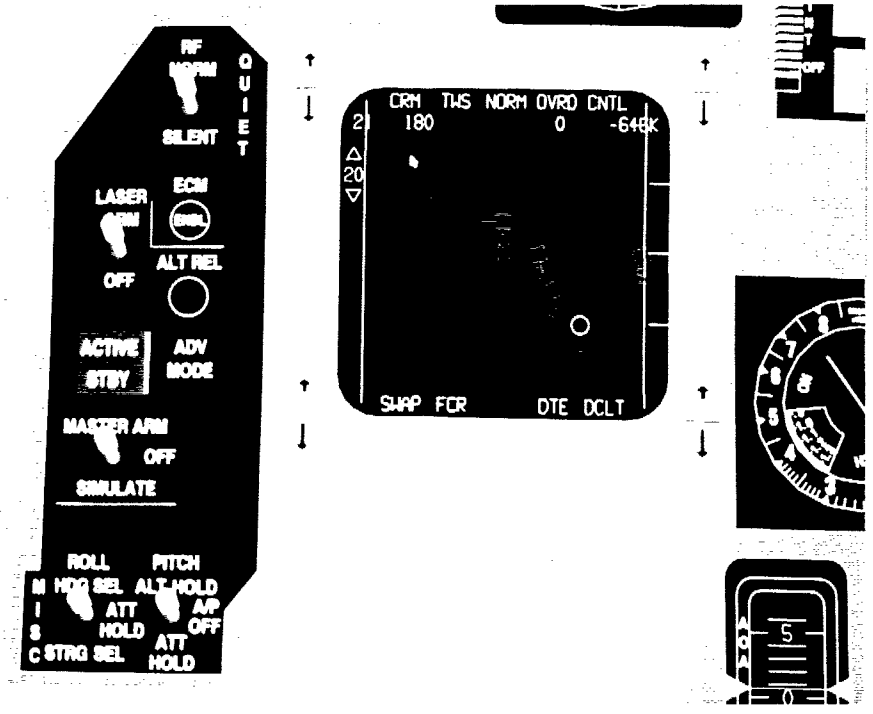

Figure A-9. Bottom Left Portion of F-16 Virtual Cockpit

Appendix B: Common Object DataBase Software Listings

This appendix contains the source code for the Common Object DataBase (CODB). The CODB is built upon the DoubleBuffer class, the header file is provided here for reference. The files include commonobjdb.h, commonobj.cc, doublebuffer.h

B.1. commonobjdb.h

```
#ifndef __CommonObjectDB__
#define __CommonObjectDB__

#include "doublebuffer.h"
#include <iostream.h>
#include <pf.h>
#include <pr.h>

//*****************************************************************
// Class:                    CommonObjectDB
//
// Functionality:            Provides a double buffering scheme in which to
//                           store data.  Structure will store any type of
//                           data and prevent multiple concurrent writes.
//                           All users in a process using the CODB will
//                           be returned a common pointer that will
//                           be used as the pointer to gain access to the class.
//
//                           Uses instance variable of type CODBStaticStuff to
//                           store the information about the doublebuffers for
//                           the numerous types of instantiations.  The static
//                           function InitializeCODB must be called at the
//                           beginning of the program before declaring any
//                           CommonObjectDB's and after pfInit which
//                           initializes Performer shared memory. (see use)
//
// Use:     Here is a sample code segment of CODB use:
//
// #include "commonobjdb.h"
// void main (int argc, char *argv[])
// {
//       //Declare a pointer to the template CommonObjectDB which
//       //will be instantiated on the structure
//       //entity_appearance_container.
//       CommonObjectDB<entity_appearance_container> *my_codb;
//
//       entity_appearance_container *dis_data;
//
//       //Initialize Performer shared memory
//       pfInit();
//
//       //Initialize CODB data structures
//       InitializeCODB();
//
//       //Instantiate the template using the CODB constructor which
//       //takes as an argument an of the enumerated type StructureType
//       //which indicates how  the entity_appearance_container
//       //structure will be referred to in the CODB (although
//       //the same structure can be stored in the CODB under different
//       //references / enumerated type values.
//       SH->my_codb = new CommonObjectDB<entity_appearance_container>(LocalCoordStruct);
//
//       dis_data   = (entity_appearance_container*)SH->my_codb->BeginRead(LocalCoordStruct);
//          // User is returned a pointer to a structure of type entity_appearance_container
//          // the structure pointed to contains the most recently written information
//          // and the user can read this data.  Given the user is given a pointer
//          // they can also manipulate the structure which violates the intent of
//          // this function and architecture.
//       SH->my_codb->EndRead(LocalCoordStruct);
//
```

```
//       dis_data   = (entity_appearance_container*)SH->my_codb->BeginReadWrite(LocalCoordStruct);
//          // User is returned a pointer to a structure of type entity_appearance_container
//          // the structure pointed to contains the most recently written information
//          // and the user can overwrite any part of the data structure.  Note:  This
//          // routine involves some copying of the data, so if the entire structure is
//          // to be overwritten use the BeginWrite/EndWrite methods discussed below.
//    SH->my_codb->EndReadWrite(LocalCoordStruct);
//
//    dis_data   = (entity_appearance_container*)SH->my_codb->ReadWrite(LocalCoordStruct);
//          // User is returned a pointer to a structure of type entity_appearance_container
//          // in which the user must overwrite the entire structure.  Any values which
//          // are stored at the pointer's location are not valid values and should
//          // not be used.  Note:  If the user wishes to overwrite the entire structure
//          // without reading it this is the quicker method.
//    SH->my_codb->EndWrite(LocalCoordStruct);
// }
//
// Author:      T. Adams and B. Zurita and D. Wells
//
// Revision History:
// 24 Jan 96   TAA
//          - Initial write 24 Jan 96
// 17 Apr 96   WDW
//          - Magic stuff added
// 25 Apr 96   VBZ
//          - Added AcftCtrlStruct, StateStruct to StructureType enumeration.
// 08 May 96   WDW
//          - Added EnvironStruct, PerformerWSMStruct to StructureType enumeration.
//          - upped the number of possible structs to 20
// 08 May 96   TAA
//          - changed BeginMagic and EndMagic to BeginReadWrite and EndReadWrite
//          - overloaded BeginReadWrite and EndReadWrite to BeginMagic and EndMagic
//            to support the other poor souls how have such a function name in
//            their software.
// 08 May 96   WDW
//          - Added EnvironStruct, PerformerWSMStruct to StructureType enumeration.
//          - upped the number of possible structs to 20
// 06 Jun 96   GEW
//          - Added PlanetStruct, MoonStruct
// 08 Jun 96   VBZ
//          - Added HLDEStruct, LLDEStruct, CDEStruct to StructureType enumeration.
// 17 Jun 96   GEW
//          - Added AsteroidStruct, CometStruct
// 18 Jun 96   GEW
//          - Added TimeStruct, TailsStruct, SatStruct, SMStruct
// 20 Jun 96   GEW
//          - Added GUIStruct
// 26 Jun 96   WDW
//          - Added WSMEntityStruct, WSMEventStruct, WSMMgtStruct
//          - Added OwnMgtStruct, OwnStateStruct  (Own = Sending Struct to DIS Mgr)
//          - Added WSMpfEventStruct, WSMpfEntityStruct
//          -----NEED TO REMOVE: WSMStruct, StateStruct, PerformerWSMStruct
// 27 Jun 96   TAA
//          - Changed Constructor method to store buffers in the static
//            array in the position of their enumerated value (st).
//            this eliminates the searching previously required.
//          - Added a new static variable codb_initialized to
//            determine when CODB was initialized.
//          - Added MAX_CODB_BUFFERS constant for number of CODB buffers
//            instead of using literal
//          - Eliminated the FindStructure Routine
//          - Changed BeginReadWrite and EndReadWrite to be able to
//            write to any type of structure.  Previously it worked
//            incorrectly because the memcpy was on the wrong size of
//            memory.  This invovled adding the size of the data type
//            into the CODB Structure.
// 19 Jul 96   VBZ
//          - Changed HLDEStruct to SDEStruct, and LLDEStruct to TDEStruct.
// 29 Jul 96   GEW
//          - Added WSMGPSStruct
// 1 Aug 96    BWG
//          - Added PatientVitals, DoctorTreatment
// 1 Aug 96    VBZ
```

```
//          - Added WingmanStruct
// 09 Aug 96, VBZ
//          - Copied entire contents of StructureType from the actual
//             CODB in use by everyone.  Steven Sheasby made some changes to
//             DIS Manager, namely, he's not using WSMStruct anymore.  Now
//             he's using WSMEntityStruct instead.
// 4 Sep 96   BWG
//          - Added Warmer, IV, Defib Structs
// 11 Sep 96   TAA
//          - Combinded the two versions of CODB into one copy
//          - added the following structures: CockpitStruct, RadarStruct,
//             INSStruct, MFDStruct, WeaponStruct
//          - Registered method to determine if a Structure has been
//             registered in the CODB (taken from VBZ's implementation)
//01 Oct 96   TAA
//          - Fixed a problem with CODB which occured when a new structure
//             was added after a program forked.  This caused both processes
//             to have an inconsistent list_of_buffers data structure.
//             This happened because the static members of the CODBStaticStuff
//             class were being allocated from free store by the compiler
//             and needed to be allocated from shared memory.  The solution
//             was to simulate static variables by ensuring that memory
//             was only allocated once for them.  This was accomplished
//             by overloading the new operator in the CODBStaticStuff class
//             to only allocate memory once for the class.  In addition, a
//             static member function was added to the CODB which could be
//             used to initialize the CODBStaticStuff class before the
//             procedures were forked.  CommonObjectDB was changed
//             to include a class member of the type CODBStaticStuff instead
//             of inheriting behavior from CODBStaticStuff.  A class new
//             and delete operator were also added.
//*****************************************************************

#define MAX_CODB_BUFFERS 50

enum StructureType {MouseStruct, KeyboardStruct, AircraftStruct, RendererStruct, HotasStruct,
                    FastrakStruct, WSMStruct, PodStruct, AcftCtrlStruct, StateStruct, EnvironStruct,
                    PerformerWSMStruct, PlanetStruct, MoonStruct, SDEStruct, TDEStruct, CDEStruct,
                    AsteroidStruct, CometStruct, TimeStruct, TrailsStruct, SMStruct, SatStruct,
                    GUIStruct, DummyStruct,
                    WSMEntityStruct, WSMEventStruct, WSMMgtStruct,
                    WSMpfEntityStruct, WSMpfEventStruct,
                    OwnStateStruct, OwnMgtStruct, WSMGPSStruct,
                    PatientVitals, DoctorTreatment, IVPumpStruct, PatientWarmerStruct, DefibStruct,
                    WingmanStruct, CockpitStruct, RadarStruct, INSStruct, MFDStruct, LocalCoordStruct,
                    WeaponStruct
};

class CODBStaticStuff
{
protected:
public:
     struct buffer_struct
     {
          StructureType type_of_structure;
          void   *ptr_to_buffer;
          int sizeof_T;
     }; //end buffer_struct

     static CODBStaticStuff *StaticCODB;
     buffer_struct list_of_buffers[MAX_CODB_BUFFERS];

public:
     void* operator new(size_t size)
     {
          if (StaticCODB == NULL)
          {
               StaticCODB = (CODBStaticStuff*)pfMalloc(size, pfGetSharedArena());
               //Initialize data structure
               for(int j=0; j<MAX_CODB_BUFFERS; j++)
```

```
            { //begin loop through buffers StaticCODB->list_of_buffers[j].ptr_to_buffer
                = NULL; StaticCODB->list_of_buffers[j].sizeof_T = 0;
            } //end if !codb_initialized
        }
        return (StaticCODB);
    };

    void operator delete(void* ptr)
    {
        pfFree(ptr);
    };
}; //end of CODBStaticStuff};

template <class T>
class CommonObjectDB
{ //begin class CommonObjectDB

public:
    CODBStaticStuff *CODB;

    CommonObjectDB(StructureType);
    void* operator new(size_t size);
    void  operator delete(void* ptr);
    int  Registered(StructureType);
    void* BeginRead(StructureType);
    void  EndRead(StructureType);
    void* BeginWrite(StructureType);
    void  EndWrite(StructureType);
    void* BeginReadWrite(StructureType);
    void  EndReadWrite(StructureType);
    void* BeginMagic(StructureType);
    void  EndMagic(StructureType);

}; //end class CommonObjectDB

static void InitializeCODB()
{
    CODBStaticStuff* x;
    x = new CODBStaticStuff;
}

#endif
```

B .2 . commonobjdb.cc

```
#include "doublebuffer.h"
#include "commonobjdb.h"
#include <iostream.h>

//************************************************************** // Class:
CommonObjectDB
//
// Functionality:      Provides a double buffering scheme in which to
//                           store data.  Structure will store any type of
//                           data and prevent multiple concurrent writes.
//                           All users in a process using the CODB will
//                           be returned a common pointer that will
//                           be used as the pointer to gain access to the class.
//
//                           Inherits from the CODBStaticStuff class which allows
//                           a template to maintain a static structure for
//                           numerous types of instantiations.
//
//
// Author:      T. Adams and B. Zurita and D. Wells
//
// Revision History:
// 24 Jan 96   TAA
//          - Initial write 24 Jan 96
// 17 Apr 96   WDW
//          - Magic stuff added
// 25 Apr 96   VBZ
//          - Added AcftCtrlStruct, StateStruct to StructureType enumeration.
// 08 May 96   WDW
//          - Added EnvironStruct, PerformerWSMStruct to StructureType enumeration.
//          - upped the number of possible structs to 20
// 08 May 96   TAA
//          - changed BeginMagic and EndMagic to BeginReadWrite and EndReadWrite
//          - overloaded BeginReadWrite and EndReadWrite to BeginMagic and EndMagic
//             to support the other poor souls who have such a function name in
//             their software.
// 08 May 96   WDW
//          - Added EnvironStruct, PerformerWSMStruct to StructureType enumeration.
//          - upped the number of possible structs to 20
// 06 Jun 96   GEW
//          - Added PlanetStruct, MoonStruct
// 08 Jun 96   VBZ
//          - Added HLDEStruct, LLDEStruct, CDEStruct to StructureType enumeration.
// 17 Jun 96   GEW
//          - Added AsteroidStruct, CometStruct
// 18 Jun 96   GEW
//          - Added TimeStruct, TailsStruct, SatStruct, SMStruct
// 20 Jun 96   GEW
//          - Added GUIStruct
// 26 Jun 96   WDW
//          - Added WSMEntityStruct, WSMEventStruct, WSMMgtStruct
//          - Added OwnMgtStruct, OwnStateStruct  (Own = Sending Struct to DIS Mgr)
//          - Added WSMpfEventStruct, WSMpfEntityStruct
//          -----NEED TO REMOVE: WSMStruct, StateStruct, PerformerWSMStruct
// 27 Jun 96   TAA
//          - Changed Constructor method to store buffers in the static
//             array in the position of their enumerated value (st).
//             this eliminates the searching previously required.
//          - Added a new static variable codb_initialized to
//             determine when CODB was initialized.
//          - Added MAX_CODB_BUFFERS constant for number of CODB buffers
//             instead of using literal
//          - Eliminated the FindStructure Routine
//          - Changed BeginReadWrite and EndReadWrite to be able to
//             write to any type of structure.  Previously it worked
//             incorrectly because the memcpy was on the wrong size of
//             memory.  This involved adding the size of the data type
//             into the CODB Structure.
```

```
// 19 Jul 96   VBZ
//          - Changed HLDEStruct to SDEStruct, and LLDEStruct to TDEStruct.
// 29 Jul 96  GEW
//          - Added WSMGPSStruct
// 1 Aug 96   BWG
//          - Added PatientVitals, DoctorTreatment
// 1 Aug 96   VBZ
//          - Added WingmanStruct
// 09 Aug 96, VBZ
//          - Copied entire contents of StructureType from the actual
//             CODB in use by everyone.  Steven Sheasby made some changes to
//             DIS Manager, namely, he's not using WSMStruct anymore.  Now
//             he's using WSMEntityStruct instead.
// 4 Sep 96   BWG
//          - Added Warmer, IV, Defib Structs
// 11 Sep 96   TAA
//          - Combinded the two versions of CODB into one copy
//          - added the following structures: CockpitStruct, RadarStruct,
//             INSStruct, MFDStruct
//          - Registered method to determine if a Structure has been
//             registered in the CODB (taken from VBZ's implementation)
//01 Oct 96   TAA
//          - Fixed a problem with CODB which occured when a new structure
//             was added after a program forked.  This caused both processes
//             to have an inconsistent list_of_buffers data structure.
//             This happened because the static members of the CODBStaticStuff
//             class were being allocated from free store by the compiler
//             and needed to be allocated from shared memory.  The solution
//             was to simulate static variables by ensuring that memory
//             was only allocated once for them.  This was accomplished
//             by overloading the new operator in the CODBStaticStuff class
//             to only allocate memory once for the class.  In addition, a
//             static member function was added to the CODB which could be
//             used to initialize the CODBStaticStuff class before the
//             procedures were forked.  CommonObjectDB was changed
//             to include a class member of the type CODBStaticStuff instead
//             of inheriting behavior from CODBStaticStuff.  A class new
//             and delete operator were also added.
//*****************************************************************
//Initialize static variables
CODBStaticStuff* CODBStaticStuff::StaticCODB = NULL;

//*****************************************************************
// Function:        Constructor
//
// Functionality:   Creates a doublebuffer class based on T and
//                       the parameter st passed into constructor.  If
//                       the same structure type is passed in the
//                       new structure is ignored.
//
// Author:          T. Adams and B. Zurita
//
// Revision History:
// 24 Jan 96   TAA
//          - Initial write 24 Jan 95
// 27 Jun 96   TAA
//          - Changed access method to store buffers in the static
//             array in the position of their enumerated value (st).
//             this eliminates the searching previously required.
//*****************************************************************
template<class T> CommonObjectDB<T>::CommonObjectDB(StructureType st)
{ //begin constructor
    CODB = new CODBStaticStuff();
    if (CODB->list_of_buffers[st].ptr_to_buffer == NULL)
    { //begin if buffer hasn't been created
        CODB->list_of_buffers[st].type_of_structure = st; CODB-
        >list_of_buffers[st].sizeof_T = sizeof(T); CODB-
        >list_of_buffers[st].ptr_to_buffer =
            (void*)(new DoubleBuffer<T>);
        cerr << "Structure " << st << " is being created." << CODB->list_of_buffers[st].ptr_to_buffer << '
    created
    else
    { //begin else buffer has been created
```

```
        cerr << "Structure " << st << " already created." << CODB->list_of_buffers[st].ptr_to_buffer << e
    created
}  //end constructor

//*****************************************************************
// Function:  operator new
//
// Functionality:  Allocates the memory for a CommonObjectDB class
//            from shared memory.
//
// Author:    T. Adams
//
// Revision History:
// 02 Oct 96   TAA
//     - Initial write.
//*****************************************************************
template<class T>
void* CommonObjectDB<T>::operator new(size_t size)
{
return( (CommonObjectDB*)pfMalloc(size, pfGetSharedArena()));
};

//*****************************************************************
// Function:  operator delete
//
// Functionality:  Frees the memory for a CommonObjectDB class. //
// Author:    T. Adams
//
// Revision History:
// 02 Oct 96   TAA
//     - Initial write.
//*****************************************************************
template<class T>
void CommonObjectDB<T>::operator delete(void* ptr)
{
    pfFree(ptr);
};

//*****************************************************************
// Function:   Registered
//
// Functionality:   Returns 0 (zero) if the structure in question
//             hasn't been registered yet.  Returns 1 if it
//             has.
//
// Author:     T. Adams and B. Zurita
//
// Revision History:
// 29 Jul 96   VBZ
//     - Initial write.
//*****************************************************************
template<class T>
int CommonObjectDB<T>::Registered(StructureType st)
{  //begin Registered
return !(CODB->list_of_buffers[st].ptr_to_buffer == NULL);
} //end Registered
//*****************************************************************
// Function:   BeginRead
//
// Functionality:             Returns the pointer to a structure that can
//                            be read by the calling class.  First finds the
//                            appropriate buffer and then calls the
//                            identical routine for the doublebuffer class.
//
//   Author:      T. Adams and B. Zurita
//
// Revision History:
// 24 Jan 95   TAA
//      - Initial write 24 Jan 95
//*****************************************************************
template<class T>
void* CommonObjectDB<T>::BeginRead(StructureType st)
```

```
{ //begin BeginRead
     DoubleBuffer<T>* temp;

temp = (DoubleBuffer<T>*)CODB->list_of_buffers[st].ptr_to_buffer;
     return(((void*)temp->BeginRead()));
} //end BeginRead

//*****************************************************************
// Function:   EndRead
//
// Functionality:            Signals doublebuffer class that calling
//                           function is done reading the doublebuffer.
//
//   Author:      T. Adams and B. Zurita
//
// Revision History:
// 24 Jan 95   TAA
//      - Initial write 24 Jan 95
//*****************************************************************
template<class T>
void CommonObjectDB<T>::EndRead(StructureType st)
{ //begin EndRead
     DoubleBuffer<T>* temp;
temp = (DoubleBuffer<T>*)CODB->list_of_buffers[st].ptr_to_buffer;
     temp->EndRead();
} //end EndRead

//*****************************************************************
// Function:   BeginWrite
//
// Functionality:            Returns the pointer to a structure that can
//                           be written to by the calling class.  First finds the
//                           appropriate buffer and then calls the
//                           identical routine for the doublebuffer class.
//
//   Author:      T. Adams and B. Zurita
//
// Revision History:
// 24 Jan 95   TAA
//      - Initial write 24 Jan 95
//*****************************************************************
template<class T>
void* CommonObjectDB<T>::BeginWrite(StructureType st)
{ //begin BeginWrite
     DoubleBuffer<T>* temp;
temp = (DoubleBuffer<T>*)CODB->list_of_buffers[st].ptr_to_buffer;
     return((void*)(temp->BeginWrite()));
} //end BeginWrite

//*****************************************************************
// Function:   EndWrite
//
// Functionality:            Signals doublebuffer class that calling
//   function is done writing to the doublebuffer.
//
//   Author:      T. Adams and B. Zurita
//
// Revision History:
// 24 Jan 95   TAA
//      - Initial write 24 Jan 95
//*****************************************************************
template<class T>
void CommonObjectDB<T>::EndWrite(StructureType st)
{  //begin EndWrite
     DoubleBuffer<T>* temp;
temp = (DoubleBuffer<T>*)CODB->list_of_buffers[st].ptr_to_buffer;
     temp->EndWrite();
}  //end EndWrite


//*****************************************************************
```

```
// Function:   BeginReadWrite
//
// Functionality:            Returns the pointer to a structure that can
//   be written to by the calling class.  First finds the
//   appropriate buffer and then copies the information
//   currently in the buffer over to the writer's location
//   so that partial updates to the buffer are possible.
//
//   Author:                 D. Wells and T. Adams
//
// Revision History:
// 17 Apr 96   WDW
//      - Initial write 17 Apr 96
// 27 Jun 96   TAA
//      - Changed BeginReadWrite and EndReadWrite to be able to
//        write to any type of structure.  Previously it worked
//        incorrectly because the memcpy was on the wrong size of
//        memory.  This invovled adding the size of the data type
//        into the CODB Structure.
//*****************************************************************
template<class T>
void* CommonObjectDB<T>::BeginReadWrite(StructureType st)
{  //begin BeginReadWrite

   void* temp;
   void* write_ptr;

    //set up the writing
    DoubleBuffer<T>* tempw;
    tempw = (DoubleBuffer<T>*)CODB->list_of_buffers[st].ptr_to_buffer; write_ptr = tempw-
    >BeginWrite();

    //set up the reading
    DoubleBuffer<T>* tempr;
    tempr = (DoubleBuffer<T>*)CODB->list_of_buffers[st].ptr_to_buffer; temp = (void
    *)(tempr->BeginRead());

    //bitwise memory copy from reader to writer
    memcpy(write_ptr, temp, CODB->list_of_buffers[st].sizeof_T);

    //close up the open read semaphore
    tempr->EndRead();

    //return the writer pointer
    return(write_ptr);

}  //end BeginReadWrite

//*****************************************************************
// Function:   EndReadWrite
//
// Functionality:   Signals doublebuffer class that calling
//                  function is done writing to the doublebuffer. //
//   Author:        D. Wells and T. Adams
//
// Revision History:
// 24 Jan 96   TAA
//      - Initial write 24 Jan 96
// 17 Apr 96   WDW
//      - Copied and renamed for use as EndReadWrite
//*****************************************************************
template<class T>
void CommonObjectDB<T>::EndReadWrite(StructureType st)
{  //begin EndReadWrite
    DoubleBuffer<T>* temp;
temp = (DoubleBuffer<T>*)CODB->list_of_buffers[st].ptr_to_buffer;
    temp->EndWrite();
}  //end EndReadWrite
```

```
//*****************************************************************
// The following functions were the ancestors of BeginReadWrite
// and EndReadWrite.  They were developed and named by D. Wells.
// T. Adams changed the functions as detailed in the comments
// and renamed the functions to  BeginReadWrite
// and EndReadWrite.
//*****************************************************************
template<class T>
void* CommonObjectDB<T>::BeginMagic(StructureType st)
{  //begin BeginReadWrite
   return(BeginReadWrite(st));
}  //end BeginMagic
//*****************************************************************
template<class T>
void CommonObjectDB<T>::EndMagic(StructureType st)
{  //begin EndMagic
     EndReadWrite(st);
}  //end EndMagic
//*****************************************************************
```

B .3 . doublebuffer.h

```
#ifndef __DoubleBuffer__
#define __DoubleBuffer__

//*****************************************************************
// Class:      DoubleBuffer
//
// Functionality:      Provides a double buffer template for any structure.
//                     Fuctions can read and write from the doublebuffers
//                     at the same time.  This is currently tested only
//                     for single processes.  This class should be allocated
//                     with a new if the double buffer is to come from
//                     shared memory (Performer Arena).
//
// Author:             T. Adams and B. Zurita
//
// Revision History:
// 17 Jan 96   TAA
//             - Initial write 17 Jan 96
// 23 Feb 96   TAA
//    - Removed static storage for buffers and replaced with dynamic
//      allocation in constructor
//  5 Mar 96 VBZ
//    - Incorporated semaphores.
//  12 Sep 96 TAA
//    - Overloaded new operator to allow direct allocation of performer
//      memory.  Previously memory was allocated off the stack and
//      allocated from performer (in fitting with how the shared memory
//      was allocated by previous students).
//    - Overloaded delete to undo what was done by new
//    - Added destructor.
//*****************************************************************

#include <ulocks.h> // Semaphore stuff.

template <class T>
class DoubleBuffer
{ //begin class DoubleBuffer
private:
   int read_count;
// 5 Mar 96   int write_count;
// 5 Mar 96   int swapping_buffers;
   T *read_ptr;
   T *write_ptr;

   // Semaphore stuff.  dummy is used as contents of shared arena.  Shared_DB
   // is the pointer to that shared arena.  The only reason for the shared
   // arena is to provide a home for the semaphores.
   int dummy;
   ulock_t WriteAccess;
   ulock_t ReadAccess;
   ulock_t RdCtAccess;
//23 Feb 96   T first_buffer;
//23 Feb 96   T second_buffer;
public:
   void *operator new(size_t);
   void operator delete(void* ptr);
   DoubleBuffer(void);
   ~DoubleBuffer(void);
   T* BeginRead(void);
   void EndRead(void);
   T* BeginWrite(void);
   void EndWrite(void);

}; //end class DoubleBuffer
#endif
```

Bibliography

[AMSE95] Amselem, Denis. "A Window on Shared Virtual Environments," *Presence* Volume 4, Number 2: 130-145 (Spring 1995).

[ANDE93] Andersson, R. "A Real Experiment in Virtual Environments: A Virtual Batting Cage," *Presence* Volume 2, Number 1: 16-33 (Winter 1993).

[ASTH93] Astheimer, P, W. Felger, and S. Muller. "Virtual Design: A generic VR system for industrial applications," *Computers & Graphics, An International Journal* Volume 17, Number 6: 671-677 (1993).

[BENS96] Zurita, B., "A Domain Independent Knowledge Based Architecture for Computer Generated Forces," Masters Thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/96D-XX, December 1996.

[BOLA94] Bolas, Mark T., "Human Factors in the Design of an Immersive Display," *IEEE Computer Graphics and Applications*. 55-59 (January 1994).

[CARL93] Carlsson, Christer and Olof Hagsand. "DIVE - A Platform for Multi-User Virtual Environments," *Computer Graphics, An International Journal* Volume 17, Number 6: 663-669 (1993).

[CATE95] Cater, Joseph, and Stephen Huffman. "Use of the Remote Access Virtual Environment Network (RAVEN) for Coordinated IVA-EVA Astronaut Training and Evaluation," *Presence* Volume 4, Number 2: 103-109 (Spring 1995).

[COOK92] Cooke, Joseph, Michael Zyda, David Pratt, and Robert McGhee. "NPSNET: Flight Simulation Dynamic Modeling Using Quaternions," *Presence* Volume 1, Number 4: 404-420 (Fall 1992).

[DIAZ93] Diaz, Milt. *The Photo-Realistic Virtual Cockpit.* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/93-07, December 1993.

[ERIC93] Erichsen, Matthew Nick. *Weapon System Sensor Integration for a DIS-Compatible Virtual Cockpit.* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/93-07, December 1993.

[FAA95] Federal Aviation Administration. Reconfigurable Cockpit Station WWW Site, http://www.tc.faa.gov/rcs/rcsdocs/adl95.html#RCS.

[FIGU93] Figueiredo, M., K. Bohm, and J. Teixeira. "Virtual Design: Advanced Interaction Techniques in Virtual Environments," *Computers & Graphics, An International Journal* Volume 17, Number 6: 655-661 (1993).

[GARC96] Garcia, B., "Design And Prototype Of The AFIT Virtual Emergency Room:A Distributed Virtual Environment For Emergency Medical Simulation," Masters Thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/96D-07, December 1996.

[GERH93] Gerhard, William Edward Jr. *Weapon System Integration for the AFIT Virtual Cockpit.* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/93-10, December 1993.

[GIVE95] Givens, Brett R. "Cockpit Display Prototyping for an Engineering Design Simulator," *Proceedings of The IEEE 1995 National Aerospace and Electronics Conference*, NAECON 1995. 722-728, May 22-26, 1995.

[GRIE96] Grier, James. F-16 Pilot, Air Force Institute of Technology, Wright-Patterson Air Force Base, OH. Personal interview. 31 October 1996.

[GUM94] Gum, Don R. "Engineering Flight Simulation, Capabilities and Future Direction for Wright Laboratory." Informational Report for Visitors to Wright Laboratory's Control Integration and Assessment Branch, WL/FIGD, Wright-Patterson Air Force Base, OH. 1994.

[HAGS96] Hagsand, Olaf. "Interactive Multiuser VEs in the DIVE System," *IEEE MultiMedia* Volume 3, Number 1: 30-39 (Spring, 1996).

[HARV91] Harvey, Edward P. and Richard L. Schaffer. "The Capability of the Distributed Interactive Simulation Network Standard to Support High Fidelity Aircraft Simulation,." *Proceedings of the Thirteenth Interservice/Industry Training Systems Conference*, Orlando, Florida, pp.127-135, 1991.

[KEST94] Kestermann, Jim B. *Immersing the User in a Virtual Environment: The AFIT Information Pod Design and Implementation.* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/94D-15, December 1994.

[IEEE93] Institute of Electrical and Electronics Engineers, International Standard, ANSI/IEEE Std 1278-1993, Standard for Information Technology, Protocols for Distributed Interactive Simulation, March 1993.

[LORA95] Loral. *ModSAF Software Architecture and Overview Document, Version 1.5.1*, Loral Advanced Distributed Simulation Technology Program Office, ADST/WDL/TR-95-W003339B, 28 April 1995.

[MACE94] Macedonia, Michael, Michael Zyda, David Pratt, Paul Barham, and Steven Zeswitz." NPSNET: A Network Software Architecture for Large-Scale Virtual Environments," *Presence* Volume 3, Number 4: 265-287 (Fall 1994).

[MCCA94] McCarty, W. Dean, Steven Sheasby, Philip Amburn, Martin Stytz, and Chip Switzer. "A Virtual Cockpit for a Distributed Interactive Simulation," *IEEE Computer Graphics and Applications.* 49-54, January 1994.

[MCLE92] McLendon, Patricia. *IRIS Performer Programming Guide.* Mountain View, California: Silicon Graphics, Inc. 1992.

[MCDO91] McDonald, L. Bruce; Christina Bouwens, Ronald Hofer, Gene Wiehagen, Karen Danisas, and James Shiflett. "Standard Protocol Data Units for Entity Information and Interaction in a Distributed Interactive Simulation," *Proceedings of the Thirteenth Interservice/Industry Training Systems Conference.* Orlando, Florida, 119-126, 1991.

[MILB95] Milbank, Ronald J. *Designer's Workbench, 3.1 reference, Second Edition.* Los Gatos, California: Coryphaeus Software Inc. December 1995.

[MOSH86] Mosher, Charles Jr, George W. Sherouse, Peter H. Mills, Kevin L. Novins, Stephen M. Pizer, Julian G. Rosenman, and Edward L. Chaney. "The Virtual Simulator", *Proceedings of 1986 Workshop on Interactive 3D Graphics.* Chapel Hill, North Carolina, 37-42, October 23-24, 1986.

[MULT94] MultiGen. *MultiGen Modeler's Guide, Revision 14.0.* San Jose, California: Software System. March 1994.

[OGDE94] Ogden Air Logistics Center. *User's Manual For The Block 25/30/32 SCU 2 AN/APG-68 Fire Control Radar OFP 7021*, Ogden Air Logistics Center, Hill Air Force Base, Utah, SCU 2 OFP 7021, 12 May 1994.

[OLSE96] Olsen, Randy. F-16 Weapon & Tactics Trainer Program, Wright-Patterson Air Force Base, OH. Personal interview. 11 April 1996.

[ROHL94] Rohlf, John and James Helman. "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics ," *Proceedings of SIGGRAPH 94.* 1-14, July 24-29 1994.

[ROY95] Roy, Trina, Carolina Cruz-Neira, and Thomas DeFanti. "Cosmic Worm in the CAVE: Steering a High-Performance Computing Applications from a Virtual Environments," *Presence* Volume 4, Number 2: 130-145 (Spring 1995).

[RUMB91] Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice Hall 1991.

[SCRI94] Scribner, Kennard III. "Distributed Interactive Simulation and the War Breaker Zen Regard Simulation, A Participant's Perspective," *Proceedings of 1994 Royal Aeronautical Society DIS Conference*, 1994.

[SCHN95] Schneider, N.., "Dynamic Transfer of Control Between Manned and Unmanned Simulation Actors," Masters Thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/95D-XX, December 1995.

[SHAW92] Shaw, Chris, Jiandong Liang, Mark Green, and Yunqi Sun. "The Decoupled Simulation Model for Virtual Reality Systems," *Proceedings of ACM Conference on Human Factors in Computing Systems*, 321-328, 3-7 May 1992.

[SHEA92] Sheasby, M., "Management of SIMNET and DIS Entities in Synthetic Environments," Masters Thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/92D-16, December 1992.

[SING95] Singhal, Sandeep, and David Cheriton. "Using a Position History-Based Protocol for Distributed Object Visualization," *Presence* Volume 4, Number 2: 1-25, (Spring 1995).

[SNYD93] Snyder, M., "ObjectSim - A Reusable Object Oriented DIS Visual Simulation," Masters Thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/93D-20, December 1993.

[STAR96] Stark, Thomas, Richard Weatherly, and Annette Wilson. "The High Level Architecture (HLA) Interface Specification And Applications Programmer's Interface." *Proceedings 14th DIS Workshop.* March 11-15, 1996.

[STUR94] Sturman, D.J., and D. Zeltzer. "A Survey of Glove-based Input," *IEEE Computer Graphics and Applications.* 30-39, (January 1994).

[STYT97] Stytz, Martin, Terry Adams, Brian Garcia, Steven Sheasby, and Brian Zurita, "Developments in Rapid Prototyping and software Architecture for Distributed Virtual Environments," IEEE Software, to appear.

[STYT95] Stytz, Martin, Steven Sheasby, and Keith Shomper. "Using Software Containers and Object-Oriented Design and Implementation to Build Distributed Virtual Environments," 1-6, 1995.

[SWIT92] Switzer, J. C., "A Synthetic Environment Flight Simulator: The AFIT Virtual Cockpit," Masters Thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/92D-17, December 1992.

[ULST96a] Ulstead, Gary. F-16 Pilot, F-16 System Program Office, Wright-Patterson Air Force Base, OH. Personal interview. 27 September 1996.

[ULST96b] Ulstead, Gary. F-16 Pilot, F-16 System Program Office, Wright-Patterson Air Force Base, OH. Personal interview. 31 October 1996.

[WELL96] Wells, W., "Collaborative Workspaces Within Distributed Virtual Environments," Masters Thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/96D-28, December 1996.

[WL96] Wright Laboratory. PCCADS WWW Site, http://www.wpafb.af.mil/flight/fcd/figp/figp1/pccads/pccads.htm.

[WILL96] Williams, G., "Solar System Modeler: A Distributed, Virtual Environment for Space Visualization and GPS Navigation," Masters Thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/96D-29, December 1996.

[ZURI96] Zurita, B., "An Architecture For Computer Generated Forces in Complex Distributed Virtual Environments," Masters Thesis, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, OH, AFIT/GCS/ENG/96D-38, December 1996.

Vita

Captain Terry A. Adams [redacted]. He graduated from Crestview High School in 1985 and entered undergraduate studies at Bowling Green State University in Bowling Green, Ohio. He graduated with a Bachelor of Science degree in Computer Science in May 1988. He was enrolled in Reserve Officers' Training Corps while in college and received his commission upon graduation on 5 May 1989.

His first assignment was as a Software Development Manager at Wright-Patterson Air Force Base. While at Wright-Patterson, he also served as a Flight Simulation Software Engineer and an Executive Officer. His second assignment was in Las Vegas, Nevada working as a Software Engineer and Software Team Manager. In May 1995, he entered the School of Engineering, Air Force Institute of Technology. His follow-on assignment is to ACC/SC at Langley Air Force Base.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Dec 96	Master's Thesis

4. TITLE AND SUBTITLE
REQUIREMENTS, DESIGN, AND DEVELOPMENT OF A RAPIDLY RECONFIGURABLE, PHOTO-REALISTIC, VIRTUAL COCKPIT PROTOTYPE

5. FUNDING NUMBERS

6. AUTHOR(S)
Captain Terry A. Adams, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2750 P Street
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/96D-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; Distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The United States Air Force uses aircraft flight simulators for pilot training and mission rehearsal. They use a variety of simulators for this task ranging with prices ranging from $400,000 to $30,000,000. These simulators have specialized hardware that restricts reuse of their components and increases maintenance costs. Air Education and Training Command wants to reduce simulators cost and improve availability to the operational commands by supporting research in virtual reality flight simulators. This thesis looks at the development of a reconfigurable virtual cockpit in a distributed virtual environment that can be used for different aircraft as well as training scenarios. The thesis effort builds on a F-15 virtual cockpit previously developed at AFIT, by creating a F-16. The Rapidly Reconfigurable Virtual Cockpit (RRVC) allows users to switch between an F-15 and F-16 in the middle of a simulation. All software models and aircraft geometry files are updated to reflect the current aircraft. The ability of a distributed virtual environment to support two unique aircraft flight simulators in a single application was encouraging. With the development of more aircraft, a single application could be provided to the operational pilot community that would support many aircraft at a fraction of the cost of today's flight simulators.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Flight Simulators, Cockpits	125
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements.***

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract	**PR** - Project
G - Grant	**TA** - Task
PE - Program Element	**WU** - Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with....; Trans. of....; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.